FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Model-Driven Framework for Heterogeneous IoT Systems

Tiago André Silva de Amorim

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Paulo José Lopes Machado Portugal

Second Supervisor: Prof. Daniel Gouveia Costa

July 28, 2025

Resumo

O desenvolvimento de cidades inteligentes orientadas por dados tem sido apoiado principalmente pelo paradigma da Internet das Coisas (IoT), com sensores e atuadores a desempenharem um papel fundamental num vasto leque de aplicações, que abrangem desde o ambiente urbano até às infraestruturas críticas. No entanto, à medida que a fragmentação da IoT se torna uma realidade devido às diversas configurações de hardware e padrões de rede, a interoperabilidade entre dispositivos IoT heterogéneos continua a ser um desafio persistente. A interoperabilidade ao nível do hardware na camada de *Edge* é alcançada através da proposta de uma abordagem abrangente, que aproveita o padrão *Web of Things* (WoT) do *World Wide Web Consortium* (W3C) como base semântica e estrutural.

É introduzida um novo *framework*, caracterizado por um conjunto de componentes de software modulares tanto para servidores como sistemas embarcados na camada *edge*. Estes componentes, em conjunto, permitem a realização de atualizações dinâmicas *over-the-air*, a integração automatica de novos dispositivos na rede e a utilização de recursos de autoidentificação remota. Ao incorporar modelos de dados baseados no WoT em dispositivos e serviços de *Edge*, a solução proposta suporta a integração semântica entre *hardware* e protocolos de comunicação heterogéneos, facilitando a interação coerente e flexível entre os vários componentes do sistema.

Também é proposta uma interface de programação de aplicações *API* formal e um modelo relacional baseado em *Thing Models* para estruturar e gerir o comportamento e as capacidades dos dispositivos embarcados. A solução é validada por meio de uma implementação de prova de conceito, a qual demonstra a viabilidade da estrutura proposta por meio de uma implementação prática que atende aos requisitos técnicos essenciais estabelecidos pelo projeto conceitual do sistema. Esta solução é primeiramente direcionada para as cidades inteligentes. No entanto, os princípios arquitetónicos e os componentes de software desenvolvidos neste trabalho são amplamente aplicáveis a outras áreas. Por exemplo, à Indústria 4.0, à agricultura inteligente e a outros sistemas ciberfísicos.

Os resultados preliminares deste trabalho foram sujeitos a um processo de revisão por pares, tendo sido aceites para apresentação na 30.ª Conferência Internacional IEEE sobre Tecnologias Emergentes e Automação Industrial (ETFA 2025). Este processo confirmou a relevância científica do trabalho, enquanto contribuição para a comunidade de IoT e sistemas ciberfísicos.

Abstract

The development of data-driven smart cities has been primarily supported by the Internet of Things (IoT) paradigm, with sensors and actuators playing a critical role in a myriad of applications. However, as IoT Fragmentation becomes a reality due to diverse hardware and networking standard settings, interoperability among heterogeneous IoT devices remains a persistent challenge. This dissertation proposes a holistic approach for achieving hardware-level interoperability at the edge layer, leveraging the World Wide Web Consortium's (W3C) *Web of Things* (WoT) standard as a semantic and architectural foundation.

A novel system framework is introduced, featuring a suite of modular software components for both edge servers and constrained end nodes. These components collectively enable dynamic over-the-air updates, automated onboarding of new devices, and remote self-identification capabilities. By embedding WoT-based data models within devices and edge services, the framework supports semantic integration across heterogeneous hardware and communication protocols, facilitating coherent and flexible interaction across system components.

A formal application programming interface (API) and a relational *Thing Model* schema are also proposed to structure and manage device behavior and affordances. The solution is validated through a proof-of-concept implementation that demonstrates the feasibility of the proposed framework through a practical implementation that fulfills the core technical requirements established by the system's conceptual design. While designed with smart cities as the primary application domain, the architectural principles and software components developed in this work are broadly applicable to areas such as Industry 4.0, smart agriculture, and other cyber-physical systems.

Preliminary results from this work have been peer-reviewed and accepted for presentation at the 30th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2025), confirming its relevance to the broader IoT and cyber-physical systems research communities.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs)[1] provide a global framework to achieve a better and more sustainable future for all. The 2030 Agenda sets out 17 interconnected goals that address pressing global challenges, including poverty, inequality, climate change, environmental degradation, and innovation.

Technological infrastructure, particularly in the realm of the Internet of Things (IoT), plays a critical enabling role across multiple SDGs. While the *CityLink IoT Framework* developed in this dissertation does not itself target a specific application domain, it is designed to be a flexible, standards-compliant, and modular foundation that can support a wide range of IoT deployments. Its architecture allows integration of constrained devices with cloud and edge computing resources, making it applicable in diverse urban, environmental, and industrial scenarios.

In this context, *CityLink* has the potential to contribute to the following Sustainable Development Goals:

SDG 6: Clean Water and Sanitation The framework may underpin water quality monitoring or leakage detection systems in water infrastructure management.

SDG 9: Industry, Innovation and Infrastructure CityLink may be used as a backbone for scalable, resilient IoT infrastructure, especially in areas requiring energy efficiency and support for heterogeneous devices.

SDG 11: Sustainable Cities and Communities CityLink can be applied to smart city solutions for participatory planning, infrastructure optimization, and environmental data collection.

SDG 12: Responsible Consumption and Production The framework may facilitate resource monitoring and accountability in systems seeking to improve consumption patterns and production efficiency.

SDG 13: Climate Action Deployed in environmental sensing networks, CityLink can enable systems that support climate adaptation and real-time feedback loops for mitigation efforts.

SDG 16: Peace, Justice and Strong Institutions CityLink can support open-data governance platforms or public infrastructure dashboards that reinforce transparency and accountability.

SDG 17: Partnerships for the Goals CityLink adheres to open standards and promotes semantic interoperability, supporting cross-border collaboration and technology transfer.

[1] United Nations SDGs. `https://sdgs.un.org` Accessed at 28-06-2025

SDG	Target	Potential Application of CityLink	Performance Indicators and Metrics
9	9.1	Deployment of digital infrastructure supporting smart services and edge-device integration	Network uptime; Protocol support; Scalability benchmarks
	9.4	Use in energy-efficient system architectures for constrained devices	Power profiles; Embedded system compatibility; Modularity
11	11.3	Enable participatory planning through sensor-based feedback systems and public data access	API access; Civic dashboard integrations
	11.5	Use in disaster monitoring networks (e.g., flood, air quality, seismic) for early warnings	Alert delivery latency; Sensor coverage
	11.6	Monitoring of urban pollutants or waste through IoT sensing infrastructure	Types and volume of data collected; Interfacing with city services
12	12.2	Integration into systems tracking energy, water, and material use in real-time	Deployment in metering systems; Update/reporting interval
	12.4	Support for data handling policies and compliance with secure formats and open vocabularies	Presence of metadata; Access control features
13	13.3	Support climate monitoring and awareness platforms by facilitating deployment of sensor networks	Types of sensors integrated; API endpoints for climate data
6	6.3	Deployment in smart water monitoring (e.g., quality, leaks) in utility networks	Sensor coverage; Real-time data transmission capabilities
16	16.6	Basis for civic technology platforms promoting transparency and accountability through open data	Availability of public endpoints; Interoperability with institutional systems
17	17.6	Promotion of global collaboration via open standards, semantic models, and protocol interoperability	Use of W3C WoT; Cross-domain model compatibility

Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus orientadores, Paulo Portugal e Daniel Costa, pelo acompanhamento, orientação e disponibilidade. O seu contributo foi fundamental para a realização deste trabalho.

Da mesma maneira, gostaria de agradecer ao João Carlos Bittencourt, pela colaboração e ajuda ao longo de toda a dissertação. Sem a sua ajuda a definir o rumo do trabalho, os resultados desta dissertação seriam certamente inferiores.

Agradeço também à minha família por todo o apoio, incentivo e por acreditarem em mim e me incentivarem a dar sempre o meu melhor ao longo destes 5 anos de curso. Sem apoio incondicional dos meus pais e do meu irmão certamente nunca teria chegado a onde estou.

Por fim, agradeço a minha namorada e a todos os meus amigos por todos os momentos de convívio, diversão, apoio e entreajuda. Foram os meus companheiros nesta jornada, sem os quais não teria sido a mesma coisa.

Tiago André Silva de Amorim

"And so, does the destination matter? Or is it the path we take? I declare that no accomplishment has substance nearly as great as the road used to achieve it. We are not creatures of destinations. It is the journey that shapes us. Our callused feet, our backs strong from carrying the weight of our travels, our eyes open with the fresh delight of experiences lived."

Brandon Sanderson, The Way of Kings

Contents

List of Figures

List of Listings

List of Tables

List of Acronyms

AI	*Artificial Intelligence*
API	*Application Programming Interface*
APP	*Application*
CRUDL	*Create, Read, Update, Delete, LIST*
CoAP	*Constrained Application Protocol*
HTTP	*Hypertext Transfer Protocol*
IETF	*Internet Engineering Task Force*
IRI	*Internationalized Resource Identifier*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
JSON-LD	*JSON for Linked Data*
LoRaWAN	*Long Range Wide Area Network*
MQTT	*Message Queuing Telemetry Transport*
NID	*Namespace Identifier*
NSS	*Namespace-specific String*
OOP	*Object-oriented Programming*
OTAU	*Over the Air Update*
OWL	*Web Ontology Language*
RDF	*Resource Description Framework*
REST	*Representational State Transfer*
RTOS	*Real-Time Operating System*
TDD	*Thing Description Directory*
TM	*Thing Model*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
URN	*Uniform Resource Name*
UUID	*Universally Unique Identifier*
W3C	*World Wide Web Consortium*
WWW	*World Wide Web*
WoT	*Web of Things*

Chapter 1

Introduction

In response to the growing rate of urbanization worldwide and the challenges posed by large population centers, smart city solutions are becoming increasingly prevalent in creating sustainable cities and communities [1]. Cornerstone technologies such as the Internet of Things (IoT), cloud-fog-edge computing models, Artificial Intelligence (AI), and Big Data analytics have been essential building blocks for enabling this new urban paradigm [2]. These technologies have typically merged into intelligent ecosystems capable of responding to complex, cross-domain problems common to smart city applications [3], [4].

Considering the development trend in this domain, advances in the aforementioned technologies and paradigms are directly related to the feasibility of smart city initiatives. They have opened new possibilities to equip cities to monitor and manage urban services more effectively, anticipate future needs, and respond proactively to potential failures and inconsistencies. As such, smart cities must operate in a closed feedback loop where what is sensed serves as real-time input for data analytics, providing the system with updated information to optimize city processes [5].

Smart cities rely on several communication networks to connect their constituent elements, which traditionally consist of passive, single-purpose sensors. To adapt to the growing needs of modern urban environments, however, there must be a shift towards adaptive smart sensor/actuator networks that take an active role in city management [2]. Deploying such infrastructure requires sophisticated data models and infrastructure capable of handling large volumes of heterogeneous information collected from various sources, tracking spatial and temporal variations of urban variables, and providing a holistic view of the system [5].

When designing smart city systems within this dynamic scenario, the lack of interoperability between devices has been a common roadblock, a problem also known as *IoT Fragmentation* [6]. It occurs when a set of smart devices is manufactured by vendors that adhere to different (and sometimes proprietary) connectivity protocols and data models, significantly limiting the number of devices that can cooperate seamlessly and leading to *vendor lock-in*. Fragmentation is also caused by IoT systems that, regardless of the underlying hardware, are not developed with extensibility and data interoperability as part of their feature set, driving up costs of large-scale installations that depend on the cooperation of multiple independent sub-systems [6]. These

limitations have reduced flexibility regarding automatic sensor/actuator identification features and remote dynamic adaptation [7].

Several projects and initiatives, such as the Web of Things (WoT), FIWARE, and oneM2M, aim to mitigate the effects of IoT Fragmentation by establishing foundational standards and unified service ecosystems. Among these, the WoT, standardized by the World Wide Web Consortium (W3C), uniquely leverages Linked Data and Semantic Web principles, enhancing interoperability through meaningful, structured data representation [8]. WoT addresses the challenge of overly permissive data modeling by employing standardized *profiles* that balance flexibility with semantic rigor. Conversely, FIWARE's loosely defined NGSI-LD [9] data models can lead to fragmentation, as variations in data structures hinder seamless integration [10], [11]. Meanwhile, oneM2M adopts a resource-oriented, non-semantic data model designed for its custom service layer [12].

Studies relying on the WoT in the smart city domain have highlighted its potential to provide a robust, flexible, and semantically coherent framework for IoT interoperability. Moreover, it has contributed to introducing adaptive features to IoT devices, such as virtual sensors and location-related data, as well as sensors specific to particular locations [8]. WoT also shows the ability to facilitate the deployment, installation, and monitoring of edge-oriented devices by embedding data models on microcontrollers for interoperability purposes [13], [14]. However, the literature still lacks solutions to address how these standards can be utilized to model and manage large-scale IoT deployments in smart cities [8].

This work presents a holistic solution for interoperability at the edge layer of IoT deployments, facilitating self-identification and remote functional adaptation. Specifically, it introduces the software components for network edge servers and end nodes that enable features such as the automatic onboarding of new devices into the network and dynamic over-the-air updates. It also introduces a WoT-based data model that captures interactions between different components and describes an Application Programming Interface (API) necessary for device interaction.

While Smart City applications are envisioned as the primary use case, with numerous references to urban environments made throughout this document, this focus does not limit the broader applicability of the proposed solutions. The architectural principles and design choices presented here are equally relevant to other domains with similar challenges, including Industry 4.0 and smart agriculture, where interoperability, modularity, and semantic integration can be beneficial.

Lastly, an early version of the architectural contributions presented in this dissertation has been peer-reviewed and accepted for presentation at the 30th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2025) The accepted paper focuses on the core framework proposal introduced in this work, providing preliminary validation of its relevance to the broader IoT and cyber-physical systems community.

The remainder of this dissertation report is organized as follows. Chapter 2 introduces fundamental concepts necessary for understanding this work. It presents the smart city domain and the architectural paradigms typically employed in these systems. It then highlights the challenges of achieving interoperability in heterogeneous environments and discusses the role of Linked Data and Semantic Web technologies as enablers of cross-domain integration.

Chapter 3 reviews the state of the art, focusing on the W3C WoT standard. It compares WoT's approach to interoperability with other initiatives such as FIWARE and oneM2M, particularly their data modeling practices and integration capabilities. The chapter also surveys relevant communication technologies and concludes with a discussion identifying the gaps this dissertation aims to fill in the current literature.

Chapter 4 introduces the proposed *CityLink* Framework, offering a high-level conceptual overview of its architecture. It describes the interactions between edge and end nodes, outlining the software components that constitute each. Special attention is given to the *Relational TM Schema*, which serves as the semantic foundation for device behavior and interoperability, and the relational model that captures the interactions between the various system participants.

Chapter 5 provides an overview of the necessary implementation steps to realize the *CityLink* Framework. It provides a walkthrough of the design requirements, including a ground-up scenario analysis that identifies the key functional characteristics of the system and a simplified development process using the various software components developed as part of this dissertation. The main goal of this chapter is to provide a practical guide for implementing the framework and its usage, bridging the gap between the conceptual design of the previous chapter and the detailed implementation analysis presented in the next chapter.

Chapter 6 documents the development and evaluation of a proof-of-concept implementation of the CityLink Framework. This includes an in-depth analysis of the functional characteristics of the developed software components for both edge nodes and constrained end nodes. The various TMs developed alongside the proof-of-concept software elements are also presented, validating the *Relational TM Schema* proposed in the previous chapter. The chapter concludes with a system-level demonstration, validating the framework's capabilities in a deployment scenario.

Finally, Chapter 7 summarizes the main contributions of this work and their significance for modern Internet of Things deployments. It also discusses the limitations encountered with the current approach, along with possible directions for future work, especially concerning security, scalability, and integration with broader smart city platforms.

Chapter 2

Background

2.1 The Smart City Domain

A smart city is an urban environment that applies various information and communication technologies to optimize resource utilization, enhance the quality of life for its residents, and promote a harmonious relationship with the surrounding ecosystems [15]. These properties make smart cities stand out from the traditional urban paradigm since they offer positive societal impacts and climate goal achievement. Smart cities are at the forefront of numerous climate-friendly technologies, closely aligning with the United Nations' SDGs for 2030. They also constitute a growing market projected to have reached a valuation of 1000bln€ by 2025 [1].

The smart city domain aggregates an array of traditionally independent fields under its name to interconnect them into an overarching smart, sustainable, and adaptive system. In the literature, the exact listing of the subdomains that fall under the scope of a smart city can vary from author to author. In their survey of smart city practices, Syed et al. [1] list the following smart city components and some of their use cases:

1. **Smart City Services**: Digitalization and optimization of municipal tasks that sustain a city's population. Some applications include using sensors for water quality monitoring and leak detection, waste collection optimization based on real-time data, and monitoring pollution levels across different city zones.

2. **Smart Energy**: Smart grids with distributed renewable energy generation at the consumer end and self-healing and self-regulating capabilities. Integration of real-time consumer power usage data to optimize power generation according to consumption predictions from prediction models trained on historical data.

3. **Smart Health**: Telemedicine services. Improved diagnosis assistance using AI. Disease prevention and tracking can be done using real-time data about people's health, using smartphones and other health trackers.

4. **Smart Homes**: The use of sensing units installed throughout a person's home provides information about the house as well as its occupants.

5. **Smart Infrastructure**: Measuring building/bridge structural state for structural health monitoring using dedicated sensors. Predictive maintenance based on collected data.

6. **Smart Transportation**: Track driver behavior and traffic patterns to supplement route planning. Public transportation tracking. Free parking spot detection systems that guide drivers to the nearest available spot.

2.2 Smart City Architecture

From an architectural standpoint, smart city solutions usually take a layered approach. Like in the previous scenario, a consensus can be found in the literature regarding smart city layers, even if concrete layer names and description details are prone to variation. Singh et al. [16] present a model commonly found in the literature composed of 4 distinct layers: the sensing layer, the transmission/networking layer, the data management layer, and finally the application layer.

As previously mentioned in the introductory chapter, the sensing layer makes up the smart city's infrastructure that directly interfaces with the physical world. End devices such as smart sensors and actuators gather information and/or act on physical quantities of interest to the system. Clusters of end devices are controlled and monitored by edge nodes that gather and transmit information to the data management layer. Edge nodes can also do simple data transformations and filtering locally and make decisions based on heuristics and/or small AI models. Moving simple decision-making and initial data pre-processing steps to the edge helps reduce the amount of raw data that must be transmitted to the rest of the network. It helps systems to scale by keeping bandwidth usage in check while also increasing the overall responsiveness and real-time capabilities of the system [5].

In the data transmission layer lie the networking technologies (Wi-Fi, Bluetooth, 4G, 5G, LoRa, etc.) and messaging protocols (HTTP, CoAP, MQTT, for instance) that enable connectivity and communication between devices. Many solutions exist from different vendors, which were created with distinct scenarios in mind. Successful IoT installations for the smart city must be able to provide flexible networking in heterogeneous scenarios where different technologies may need to be integrated as part of the same cohesive system.

The data management layer is an intermediary between the application and sensing layers, taking advantage of transmission layer technologies for communication with both. The administration of a smart city depends on the ability to process, interpret, and associate vast quantities of data while keeping it relevant and up-to-date with the changes in the system. Here, the data models are central to database design and handling historical and real-time data. As discussed in the introduction, effective smart city solutions must enforce a consistent data model across the system so that the sensing and data management layers can seamlessly communicate and operate as a closed feedback loop. Other authors, like Syed et al. [1], further divide data management into middleware (for databases and APIs) and business layers (for data analytics and optimization), but the core idea remains the same.

At the top level of the layered design, the applications layer focuses on data presentation. It uses the collected data and insights to cater to citizens and system administrators with various services and insights. Smart city administrators can interact with the underlying system through the interfaces provided at this level.

2.3 Interoperability for Smart City Applications

Interoperability in smart cities refers to the ability of various subsystems (energy, water, transportation, waste management) to exchange information meaningfully [5]. Current solutions operate in vertical compartments because of vendor-specific hardware solutions and proprietary protocols, effectively locking clients to the ecosystem provided by individual vendors. While this may be a profitable approach for hardware and service providers, it limits the acceptability of IoT solutions, especially for applications that benefit from various participating devices.

To combat this scenario and achieve horizontal cooperation across the IoT, two types of interoperability must be considered: technical and semantic interoperability [5]. Technical interoperability is achieved through the use of common standards. It allows seamless communication between devices and ensures software is portable and adaptable for different machines. Adopting standard messaging protocols is another way to enforce technical interoperability, as they establish uniform rules for message exchange between devices.

Semantic interoperability, on the other hand, is the way different systems interpret data and how shared vocabulary can be used to enforce consistency in the meaning conveyed by messages between machines. The problem is to provide certainty that different applications attribute the same meaning to the same set of attributes or entities.

To exemplify this problem, a hypothetical smart city application with two separate IoT installations will be considered for the remainder of this section. The first is an environmental sensing unit, while the second monitors a water reservoir. Both installations publish JSON data to the cloud. Listings 2.1 and 2.2 provide example JSON payloads for the scenarios described. Although the JSON keys *temperature* and *pressure* are common to each example, their context and meaning differ. As listing 2.1 is associated with environmental measurements, *temperature* and *pressure* could more explicitly be expressed as *ambientTemperature*, measured in degrees Celsius, for instance, and *ambientPressure* measured in bar. On the other hand, the attributes in listing 2.2 could be *waterTemperature* in Celsius and *waterPressure* in psi.

Listing 2.1: simple environmental monitoring example

```
{
    "humidity": 0.7,
    "temperature": 20,
    "pressure": 1.01
}
```

Listing 2.2: simple water reservoir monitoring example

```
{
    "phLevel": 7,
    "temperature": 4,
    "pressure": 14
}
```

Although this example represents a very simplistic scenario, it stresses how differences in context necessitate semantic interoperability.

To have these IoT installations share data between themselves or to a common database, a context translation layer would need to be created to ensure the uniqueness of JSON keys so that the system does not misinterpret their respective values. Manually resolving such context translation issues is simple when integrating straightforward systems with few conflicting parameters, but quickly becomes impractical and unscalable in large smart city deployments, where dozens of different systems may need to cooperate automatically.

Automated approaches to semantic mapping, supported by ontologies and shared standards, are essential for enabling scalable and meaningful integration of heterogeneous IoT systems.

2.4 Linked Data & the Semantic Web Paradigm

Understanding the concept and practical workings of the Linked Data paradigm is essential to achieving semantic interoperability in the Internet of Things (IoT). Linked Data forms a core component of the Semantic Web, a vision proposed by Tim Berners-Lee for a Web of data that machines can interpret, share, and reason over autonomously. This section outlines key Semantic Web principles and technologies that underpin this paradigm and serve as a foundation for this dissertation.

At the heart of the Semantic Web is the **Resource Description Framework** (RDF) [17], a W3C Recommendation that provides a standard model for representing structured data. RDF encodes information as a graph of statements, each composed of a *Subject*, a *Predicate*, and an *Object*, collectively referred to as an *RDF Triple*. These triples describe relationships between entities and form the basis of an *RDF Graph*, as illustrated in Figure 2.1.

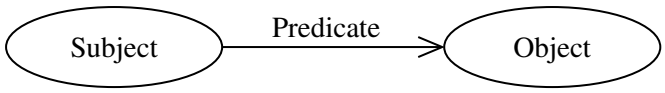


Figure 2.1: Graph representation of an RDF Triple [17]

A valid RDF Triple adheres to the following structure:

- The Subject must be an IRI [18] or a blank node.
- The Predicate must be an IRI.
- The Object may be an IRI, a literal[1], or a blank node.

RDF enables systems to describe data and relationships in a semantic, machine-readable form. Importantly, these descriptions are globally resolvable using IRIs (Internationalized Resource Identifiers), allowing machines to follow links between datasets, an essential characteristic of Linked Data.

[1] A value with an associated data type, such as an integer or a character string

Several syntaxes have been developed to serialize RDF data in a web-friendly way. Among them, JSON-LD has gained widespread adoption, particularly in web contexts. JSON-LD (JavaScript Object Notation for Linked Data) is a W3C Recommendation [19] that extends standard JSON with features that support RDF-compatible linked data.

JSON-LD maintains the simplicity and human-readability of JSON while introducing special keywords (prefixed with @) to define semantic context. These keywords automatically allow data consumers to map JSON keys to IRIs defined in external vocabularies. This approach supports backward compatibility with existing JSON systems while enabling semantic data enrichment.

```
{
    "@context": "https://json-ld.org/contexts/person.jsonld",
    "@id": "http://dbpedia.org/resource/John_Lennon",
    "name": "John Lennon",
    "born": "1940-10-09",
    "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

Listing 2.3: Example JSON-LD snippet. Taken from `https://json-ld.org/`

The Semantic Web facilitates machine-readable, interoperable data exchange across heterogeneous systems by leveraging RDF, JSON-LD, and related technologies. These foundations are especially relevant in the IoT domain, where devices and services must interpret and integrate data from diverse sources. Consequently, the Semantic Web paradigm, grounded in Linked Data principles, offers a robust approach to achieving semantic interoperability in distributed, data-intensive environments like those targeted in this work.

2.4.1 IRI, URI, URL, and URN

The terminology surrounding the various types of identifiers used in the web and the linked data paradigm can be confusing. In RFC 3987 [18], the Network Working Group of the Internet Engineering Task Force (IETF) defines the Internationalized Resource Identifier (IRI) as a generalization of the Uniform Resource Identifier (URI) [20] that allows for a broader range of characters, including those from non-Latin scripts.

The Uniform Resource Identifier (URI) is defined in RFC 3986 [20] as a compact string of characters that identifies an abstract or physical resource. IRIs are a superset of URIs since the latter are constrained to the US-ASCII character set, while IRIs can contain characters from the Universal Character Set (Unicode/ISO 10646).

The key concept here, common to both IRIs and URIs, is that they are identifiers. Linked that systems use IRIs to identify resources, which can be anything from a person to a place, an event, or even a concept. In a traditional database system, IRIs would be equivalent to the primary key of a table, uniquely identifying a record.

Also defined in RFC 3986, URLs (Uniform Resource Locators) are a specific type of URI, that, in addition to identifying a resource, also provide a means of locating it by describing its primary access mechanism, such as the protocol to use (e.g., HTTP, FTP) and the address of the resource. This means all URLs are URIs, but not all are URLs.

Finally, URNs (Uniform Resource Names) are another type of URI that exist under the *urn* scheme, as defined in RFC 2141 [21]. URNs are intended to serve as persistent, location-independent resource identifiers. URNs follow a strict syntax, being composed of a namespace identifier (NID) and a namespace-specific string (NSS), separated by a colon, following the pattern `urn:<NID>:<NSS>`.

In the context of this work, a good example of a URN is the Universally Unique Identifier (UUID) [22], a 128-bit string commonly used for identifying resources in distributed systems. Version 4 UUIDs, in particular [23], are randomly generated UUIDs often used as IDs in databases and other systems, including the work presented in this dissertation.

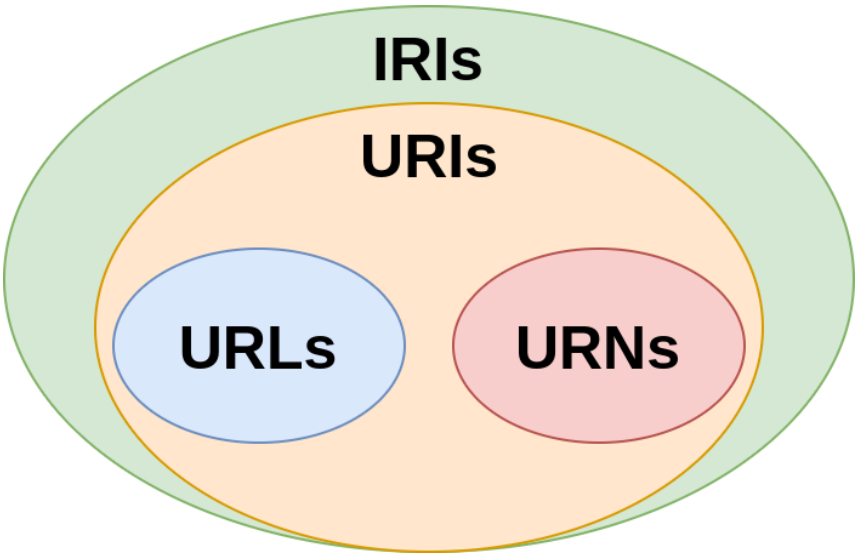


Figure 2.2: Resourse Identifier Hierarchy

Figure 2.2 illustrates the relationship between IRIs, URIs, URLs, and URNs. For this dissertation, the terms IRI and URI will be used interchangeably, as the distinction is not relevant to the work presented here; however, it is essential to keep in mind the differences between these terms, as they are often used in the literature and the context of linked data. The interchangeability of these terms stems from the fact that various standards and specifications, including the W3C's RDF and JSON-LD, use the term IRI to refer to resource identifiers. In contrast, the term URI is more commonly used in the context of web technologies and protocols. A mapping is defined from IRIs to URIs, which allows for the conversion of IRIs to URIs by percent-encoding non-ASCII characters, as per the rules described in RFC 3986 [20], meaning that IRIs can be used as URIs in most cases, and vice versa.

2.4.2 JSON-LD & RDF

JSON-LD builds on RDF by providing a more user-friendly serialization format. As mentioned above, the *@context* key instructs the JSON-LD preprocessor to convert a JSON-LD document into its expanded version and then process it as an RDF graph. The context is used to map terms to IRIs. A term is any case-sensitive string that does not contain forbidden characters as per the specification and is not a reserved JSON-LD keyword. In practice, terms are regular JSON keys, such as *name*, *born*, and *spouse* in listing 2.3 that are then transformed into IRIs by the JSON-LD preprocessor using the context provided in the *@context* key. The context defines shortcuts that allow for quick and accurate communication between involved parties and allows for preserving the simple, human-readable syntax.

Listing 2.4 shows the expanded version of the JSON-LD document in listing 2.3. Here, the JSON keys have been expanded into full IRIs, and the values have been typed according to their data type. This expanded version can then be serialized as RDF Triples, as shown in listing 2.5.

```
[
  {
    "@id": "http://dbpedia.org/resource/John_Lennon",
    "http://schema.org/birthDate": [ {
        "@type": "http://www.w3.org/2001/XMLSchema#date",
        "@value": "1940-10-09"
      } ],
    "http://xmlns.com/foaf/0.1/name": [{ "@value": "John Lennon" }],
    "http://schema.org/spouse": [{ "@id":
     ↪  "http://dbpedia.org/resource/Cynthia_Lennon" }]
  }
]
```

Listing 2.4: expanded JSON-LD representation

```
<http://dbpedia.org/resource/John_Lennon>
 ↪  <http://schema.org/birthDate>
 ↪  "1940-10-09"^^<http://www.w3.org/2001/XMLSchema#date> .
<http://dbpedia.org/resource/John_Lennon> <http://schema.org/spouse>
 ↪  <http://dbpedia.org/resource/Cynthia_Lennon> .
<http://dbpedia.org/resource/John_Lennon>
 ↪  <http://xmlns.com/foaf/0.1/name> "John Lennon" .
```

Listing 2.5: RDF Triples

These RDF Triples can be stored in an RDF database, such as the Apache Jena Triple Store Database [24]. Other database systems, such as FlureeDB [25] support JSON-LD natively, handling the expansion and conversion of JSON-LD into RDF Triples internally.

These systems enable storage and retrieval of linked data in a human-readable and machine-processable way. Linked data carries inherent meaning, as the IRIs used to identify resources are globally unique and, if they can be resolved into a web resource (i.e., a URL), they can provide additional information about the resource, such as its description, properties, and relationships with other resources. This creates a graph of interconnected resources that can be traversed and queried using standard protocols and languages, such as HTTP and SPARQL [26], respectively.

SPARQL is a query language for RDF data, standardized by the W3C, allowing complex queries to be executed against RDF graphs. This enables the retrieval of specific information from linked data, such as finding all resources related to a given resource or retrieving all resources that match a particular pattern. Various use cases for SPARQL queries and linked data in general can be found in [27], published by the W3C, including examples such as monitoring news events and avoiding traffic jams.

Finally, it should be explicitly stated how linked data, and JSON-LD in particular, can be used to resolve the semantic interoperability issues presented in section 2.3. By attaching a context to regular JSON data, it is possible to trivially avoid key conflicts between different systems, as, upon expansion, all keys are transformed into IRIs that are globally unique and attributed to the correct, application-specific data types. Listings 2.6 and 2.7 show how context can be used to facilitate interoperability for the section 2.3 example.

Listing 2.6: environmental monitoring example, with context

```
{
    "@context":
    ↪ "https://222.example.org
    ↪ /env-monitor.jsonld",
    "humidity": 0.7,
    "temperature": 20,
    "pressure": 1.01
}
```

Listing 2.7: simple water reservoir monitoring example, with context

```
{
    "@context": "https://www.exa
    ↪ mple.org/water.jsonld",
    "phLevel": 7,
    "temperature": 4,
    "pressure": 14
}
```

2.4.3 Web Ontology Language (OWL)

The Web Ontology Language (OWL) [28] is a W3C Recommendation that builds on RDF and provides a formal language for defining ontologies. An ontology formally represents a set of concepts within a domain and the relationships between those concepts. In the context of linked data, web ontologies are used to define the semantics of the data, providing a shared vocabulary and a set of rules for interpreting the data.

The W3C defines ontologies as a formalized vocabulary of terms, often covering a specific domain and shared by a community of users [29]. Once defined and applied to a set of data, these ontologies enable the creation of a shared understanding of the data, allowing for meaningful data exchange and integration across different systems.

Using examples from the previous section, the contexts used in listings 2.6 and 2.7 are presented as hypothetical files that could be used to solve the semantic interoperability problem. Taking the next step, each context could be based on an ontology defined with the specific domain in mind. For instance, the environmental monitoring context could be based on an ontology that defines the concepts of *temperature*, *humidity*, and *pressure* in the context of environmental monitoring. In contrast, the water reservoir context could be based on an ontology that defines the concepts of *water temperature*, *water pressure*, and *pH level*. These could then be derived from a more general ontology that defines the concepts of *temperature*, *pressure*, and *pH level* in a more general context, allowing for the reuse of existing ontologies and the creation of new ones that build on top of existing ones.

This way, while having semantic interoperability and well-defined contexts, the underlying relationships between the ontology concepts establish a shared understanding of the data and create linked data connections between different systems.

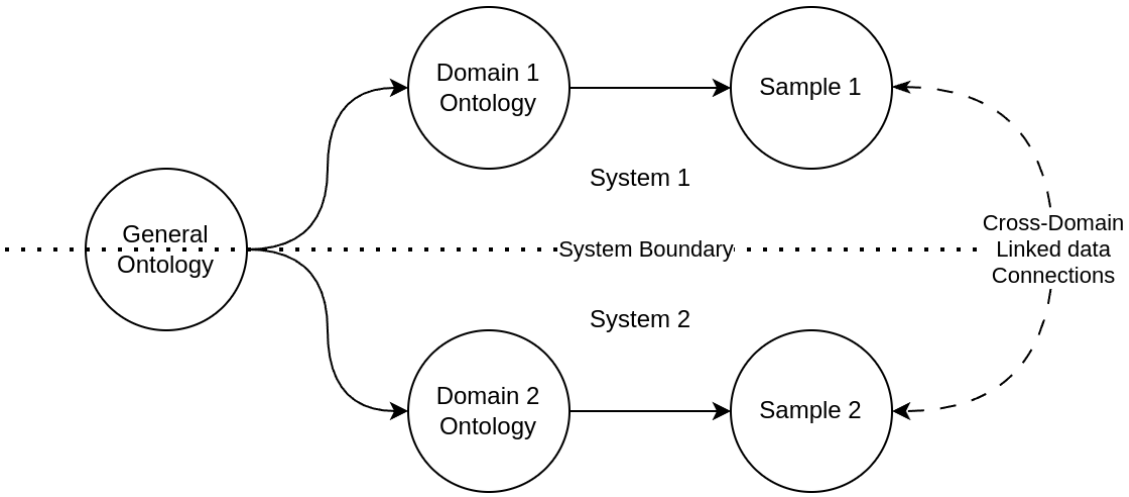


Figure 2.3: Web ontology conceptual graph representation

Figure 2.3 illustrates how linked data, enriched via web ontologies, can create implicit cross-domain connections between different systems. Looking back at the example, the environmental monitoring system (system 1) and the water reservoir monitoring system (system 2), now implemented with linked data concepts and enriched with vocabularies drawn from web ontologies, are implicitly interconnected, despite being independent systems.

Chapter 3

State of the Art

The primary focus of this dissertation work is data interoperability and its importance for IoT networks, particularly within Smart City Environments, where this problem is exacerbated due to the heterogeneous nature of modern urban centers [5]. As such, the discussion in this Chapter is mainly focused on this aspect. Two major initiatives towards the harmonization of data across the IoT will be presented: the Web of Things family of standards [30]–[32], maintained by the World Wide Web Consortium (W3C), and the NGSI-LD standard from FIWARE [9]. These initiatives build on the linked data paradigm introduced in Chapter 2 to facilitate seamless and meaningful data exchange across applications. Additionally, the non-linked data oneM2M standard [12] will be briefly addressed, providing insight into alternative technologies to the ones adopted for this dissertation work.

In the final sections of this Chapter, the discussion will pivot towards some technologies essential to the IoT, particularly networking technologies and protocols. Although not the focus of this dissertation, these technologies are ubiquitous in the Internet of Things and Smart Cities and are worthy of mention.

Finally, in Section 3.5, a summary of the technologies and works exposed will be given, reflecting on their strong points and shortcomings concerning the work developed for this dissertation.

3.1 W3C Web of Things

First proposed in 2014, the W3C Web of Things is built on the concepts of *Thing*, *Thing Description*, and *Thing Model* (TM). According to the specifications [30], [31], a *Thing* is an abstraction of a physical or virtual entity that provides intrinsic value to the IoT System and is described by standardized metadata, the *Thing Description*. TMs are optional for WoT systems and offer technological building blocks for templated instantiation of *Thing Descriptions* based on shared features.

In layman's terms, *Things* are digital representations of devices, services, or logical entities that participate in an Internet of Things ecosystem. These may include sensors, actuators, smart appliances, wearables, or cloud-based services. A *Thing Description* serves as the standardized

digital interface to a *Thing*, capturing essential metadata that enables users (denominated as *consumers* in the WoT terminology) to interact with it. *Thing Descriptions* can be considered a digital manual, or guide, on how a *Thing* can be used. They express the interaction capabilities of *Things* as what the W3C calls *Interaction Affordances*, which can be any of the following types:

- **Properties:** These can represent configuration settings, operation characteristics, or some other intrinsic quality of a *Thing*. A temperature sensor may provide its measurements as a property updated with each sample. Conversely, it may also have a property that allows configuring the sensors' sampling rate and sensitivity.
- **Actions:** As their name suggests, action affordances represent tasks or operations the *Thing* can perform. These can range from physical processes, such as actuating a valve, flipping a switch, or activating an alarm, to software tasks such as processing data or interacting with a database.
- **Events:** The third and final type of interaction affordance is primarily meant for asynchronous notifications. Events help express irregular conditions or sporadic activities that consumers need to know about. A motion detection sensor, for example, may express its detections as events. Machines may emit events when a malfunction or error is detected. Finally, software components can use events to notify consumers of updates or new data.

This property-action-event paradigm, expressed via *Thing Descriptions*, enables a machine-readable and interoperable information model that facilitates automated discovery, integration, and orchestration across heterogeneous platforms. The *CityLink* framework, developed as part of this dissertation work and presented in detail in Chapter 4 builds on the W3C Web of Things standards, taking advantage of this feature set to create a comprehensive data model for the orchestration and management of [Smart City] IoT deployments.

Listing 3.1 contains a sample for an HTTP-based *Thing Description* taken from the specification [31]. This simple example illustrates all the affordance types mentioned above, coupled with example API endpoints for the HTTP protocol. Also represented in the listing is the security aspect of the WoT standard. Security features can be directly represented in WoT documents with various protocols and access control mechanisms being supported. This is, however, outside of the scope of this dissertation, so it will not be addressed in detail.

In contrast to *Thing Descriptions*, TMs are reusable, abstract blueprints that describe common structural and behavioral characteristics shared by multiple *Things*. They do not include deployment-specific information such as concrete network addresses, but instead focus on capturing the domain-level patterns or roles. By leveraging TMs, developers and system architects can streamline the creation of *Thing Descriptions* through inheritance and specialization, promoting the consistency, modularity, and reutilization of data across WoT-based applications.

This is precisely the approach taken by this dissertation work. Where *Thing Descriptions* can be compared to objects in Object Oriented Programming (OOP), *Thing Models* make up the class definitions from which *Thing Descriptions* can be instantiated. After a TM has been defined to

for an HTTP smart lamp

```
{
    "@context": "https://www.w3.org/2022/wot/td/v1.1",
    "id": "urn:uuid:0804d572-cce8-422a-bb7c-4412fcd56f06",
    "title": "MyLampThing",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in": "header"}
    },
    "security": "basic_sc",
    "properties": {
        "status": {
            "type": "string",
            "forms": [{"href": "https://mylamp.example.com/status"}]
        }
    },
    "actions": {
        "toggle": {
            "forms": [{"href": "https://mylamp.example.com/toggle"}]
        }
    },
    "events": {
        "overheating": {
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

Listing 3.1: Thing Description sample [31]

express the general interaction affordances available to a type of device, *Thing Descriptions* can then derived from it, on a per-device basis, filling in the missing communication endpoints and additional metadata necessary to access and invoke the defined interaction affordances (such as reading properties, triggering actions, or subscribing to events) within a specific network context and protocol binding.

As a W3C Recommendation for the IoT, the WoT standard has experienced gradual adoption in both industry and academia. In [33], the authors discuss the state of IoT interoperability using the WoT, describing it as a generic, powerful mechanism and machine-readable format for representing a wide variety of devices that may connect over different protocols. This perspective aligns with the findings in [8], which identifies 11 major WoT deployments from 2017 to 2022 in academic and industrial contexts.

As members of the W3C WoT Working Group, the authors of [33] address some of the shortcomings of the WoT specifications, in particular regarding the loose normative requirements the standard places regarding concrete system architectures and modeling practices. Providing tools alone is insufficient for promoting interoperability if the way those same tools are employed is not sufficiently defined. In the face of this, the authors introduce the WoT Profile Specification [34], a mechanism meant to enable out-of-the-box interoperability by defining a set of constraints and rules regarding protocol bindings used in WoT documents. The goal for WoT profiles is to provide a normative reference, missing from the other WoT standards, to represent communication endpoints using primarily Web technologies and other adjacent protocols. A basic profile is introduced for the HTTP protocol, along with a generic *Profiling Mechanism* composed of guidelines on creating additional profiles. Profiles are composed of *Protocol Bindings*, translating WoT operation (*readproperty*, *invokeaction*, *subscribeevent*, etc) into concrete protocol-specific operations *(HTTP GET/POST*, *MQTT SUBSCRIBE/PUBLISH*, etc).

As part of the works identified in [8], six fall under the broader Smart City scope (Section 2.1), targeting the Smart Home, Environmental Monitoring, Smart Infrastructure, and Automotive (Smart Transportation) domains. The remaining applications are geared towards Industry and Agriculture.

The study in [14] leverages the WoT to retrofit pre-existing building energy management systems into an intelligent cloud-based system. Their case study is framed around the Heart Project [35], a European initiative to transform existing buildings into smart buildings to contribute to Europe's decarbonization. The WoT deployment described by the authors is executed with small single-board computers (SBCs) that act as gateway devices that manage and control machines for which they host the WoT Thing Description. The individual machine gateways then communicate with a central building gateway controller (also an SBC), which acts as an offline controller should the system lose connection to the cloud.

Smart Santander [36], [37], another European research initiative carried out in 2012, presented real-world IoT TestBeds deployed in various European cities, with large-scale sensor infrastructure. Although developed with legacy WoT technology (before the W3C standardization efforts), this initiative can serve as a baseline comparison for modern IoT standards and frameworks like the one proposed in this dissertation work. New testbeds could be created for this purpose, with older ones potentially being retrofitted with modern technologies.

As pointed out in [8], the authors of [38] presented a small-scale demonstration testbed that applied the W3C WoT standards to a multi-sensor IoT system for environmental monitoring. The authors developed sensing units equipped with eight distinct sensors: temperature, humidity, lighting, carbon dioxide, sound, organic compounds, microwave, and IrDA. These devices were deployed in a controlled environment to assess the feasibility of using the WoT family of specifications — including *Thing Descriptions* and the Scripting API — at a time when these standards were still under development [30], [31]. Sensor data were semantically modeled using Thing Descriptions and exposed through web-based APIs implemented with the Node-WoT library [39]. The demonstration verified successful interoperability, with external clients able to

query sensor readings, configure event thresholds, and receive real-time event notifications via the WoT framework.

Targeting low-power devices, the work presented in [13] proposed an Arduino-based framework for developing WoT applications for microcontrollers. This solution focused on directly hosting all the WoT functionality on the constrained device, eliminating the need for intermediaries. The proposed framework, denominated Web of Things on the Extreme Edge (WoTTEE), provides a complete vertical solution for the IoT arranged in a microservice architecture that offers cloud-deployable services and edge layer functionality.

At the cloud level, *WoTTEE* sports a graphical user interface (GUI) for project creation and sketch modification and a dashboard for data visualization and statistics backed up by a database service and other back-end functionality. The *Servient Builder* service generates device-specific project templates as part of the project creation step. These templates make it easier to develop WoT applications for microcontrollers, as the developer can focus on the application's business logic without worrying about the data model and communication code (granted that a template exists that fits the use-case requirements).

The *Microcontroller Engine* service is employed at the edge for project compilation and upload to microcontrollers. This compilation and upload step depends on Arduino-specific tools and functionality to interact with the target boards, which, in turn, limits the devices able to be used with this framework to only those that are already compatible with the Arduino ecosystem.

3.2 FIWARE NGSI-LD & Smart Data Models

The FIWARE Foundation [40] provides a curated framework of open-source components to accelerate the development of smart solutions across various domains. Under the FIWARE banner, *NGSI-LD* [9] is another linked-data oriented standard aimed towards data interoperability in the IoT. The FIWARE Smart Data Models Initiative complements this standard [41], by outlining various relational models for the digital representation of different physical entities. These do not necessarily have to be used in the context of IoT applications and are intended and generalized linked data models to be used within their respective domains. In the Smart Cities domain, specifically, these include models for Smart Buildings, Smart Streetlighting, and Smart Parking, among others.

The basis for the FIWARE data models is the *context information management framework*, denominated as NSGI-LD and standardized by the European Telecommunications Standards Institute (ETSI) [9].

NGSI-LD, like the W3C Web of Things, adopts JSON-LD as its serialization format. However, it aligns more closely with the Resource Description Framework (RDF) on which it is conceptually based. As discussed in Section 2.4, RDF structures data using triples composed of a *subject*, *predicate*, and *object*. Similarly, *NGSI-LD* organizes data around three core concepts: *Entities*, *Properties*, and *Relationships*.

An *Entity* in *NGSI-LD* represents a physical or abstract object and is described using multiple properties. *Properties* are expressed as key-value attribute pairs (e.g., `"key": "value"`), possibly including metadata such as units or observation timestamps. *Relationships* define logical links between entities and are expressed using IRIs to establish semantic meaning.

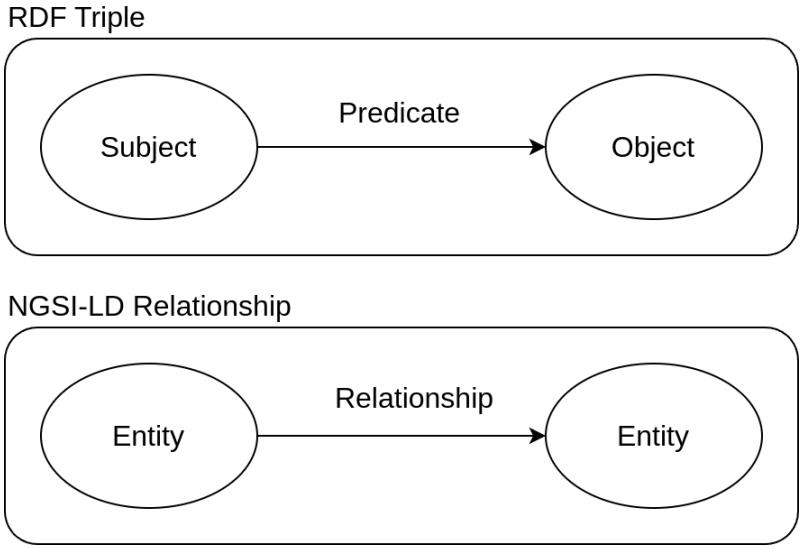


Figure 3.1: Comparison between RDF Triples and *NGSI-LD* relationships

Figure 3.1 illustrates the structural similarity between RDF triples and *NGSI-LD* data models. The key distinction lies in the *Properties* component. Whereas RDF subjects and objects are IRIs, *NGSI-LD* entities are JSON-LD documents that encapsulate richer contextual data via properties.

To standardize entity modeling across domains, the Smart Data Models initiative provides extensible, domain-specific templates for *NGSI-LD* entities. These models are designed to be modular. Users may adopt only the relevant fields to describe their use case, avoiding unnecessary data transmission overhead.

The FIWARE ecosystem supports *NGSI-LD* through a set of open-source components known as Generic Enablers [42]. These include context brokers tailored for various application domains, persistence and query services, visualization tools, and other supporting infrastructure, forming a comprehensive platform for building smart solutions. At the time of writing, however, compliance with Smart Data Models is not strictly enforced by the FIWARE context brokers. This makes using Smart Data Models optional and loosely coupled with the broader FIWARE architecture.

In the literature, several works can be found that experiment with the options provided by FIWARE. [10] used NGSI-LD to model the VALLPASS project, an Intelligent Active Surveillance (system) with LoRa Support for Pedestrian Crossings. For this work, they took advantage of several smart data models to capture the elements of the system. They created a remote management platform for the service, powered by various solutions such as FIWARE generic enablers.

The system consists of an energy-autonomous smart streetlight installed near pedestrian crosswalks that do not have traffic light signaling. These smart lampposts are made to light up when pedestrians are detected at the crosswalk to alert oncoming traffic to the presence of pedestrians.

The authors of [10] also note that despite their effort to try to find the most compatible baseline models for their use case, the pre-existing models by FIWARE did not entirely fit the system's needs in some cases and had to be amended with extra fields.

From a different perspective, the authors of [43] proposed an extension of the existing FIWARE models to support virtual devices in a *Function as a Service* deployment. They aim to provide an abstraction over physical devices and isolated services for consumers utilizing virtualization technologies on edge devices. For instance, an edge device with a temperature sensor installed may host several virtual devices, one that provides raw sensor data, another a daily average, another a weekly average, and so on. This way, a singular edge device can offer various self-contained microservices to the network that multiple users with different aims can use.

In a follow-up publication [11], the researchers leverage the FIWARE standard to propose a system architecture capable of automatically deploying and executing microservices on edge devices. Their system abstracts away physical sensors in the manner described in their previous work. It can adjust the functions executed by edge devices using NGSI-LD documents to identify services required by users and manage their life cycle. Their proof-of-concept experiments include using a Raspberry Pi 4 Model B as the edge device where the microservices are deployed.

3.3 oneM2M

The oneM2M standard [12] also addresses the challenge of IoT fragmentation with its resource-oriented architecture that does not derive from the linked data paradigm.

oneM2M's value proposition lies in its ability to enable interoperability across heterogeneous IoT components and systems through a standardized service layer. By abstracting core functionalities such as device discovery, data management, security, and communication, oneM2M facilitates the integration of devices and applications from multiple vendors without requiring custom-made interfaces or middleware. This approach reduces development and integration costs while mitigating vendor lock-in, making onM2M akin to a distributed operating system for the IoT.

[44] implements a system architecture that virtualizes common service functions to provide IoT services on the edge nodes close to the end users. Their work, supported by the oneM2M ecosystem, demonstrates how modularizing and deploying oneM2M-compliant Common Service Functions as microservices at the network edge can significantly improve latency and scalability. By aligning with oneM2M's resource-oriented architecture, the proposed framework enables dynamic slicing of IoT services and dynamic offloading of service tasks from the cloud to Multi-access Edge Computing nodes. This results in more responsive, location-aware services without compromising interoperability or standard compliance. Their prototype implementation and performance evaluation underscore the effectiveness of combining edge computing with oneM2M standards for supporting mission-critical and delay-sensitive IoT applications.

3.4 Networking Technologies

Networking is central to Smart City applications, which depend on aggregating data collected across large regions. Datasets encoded with geospatial information provide localized and contextualized information essential to address urban challenges effectively [15].

Therefore, a city-wide network capable of handling such tasks must be constructed to continuously flow information from the data sources to the data aggregation centers. Despite the advantages cloud computing provides, with its inherent scalability, flexibility, and vast computational capacity, the large scale of urban sensor networks makes it impractical to rely solely on the cloud. As the amount of information being generated increases, computational capabilities must be offloaded to devices capable of transforming and filtering raw data and taking simple data-driven decisions on-site before transmitting relevant information upstream [45].

This push to decentralize the Smart City IoT favors networks with star topologies at the edge. The connections between end devices (denominated as end nodes throughout this dissertation work) that collect information or perform tasks out in the field – a smart streetlamp or a temperature sensor for instance – to the local hubs, the edge nodes capable of handling this data, naturally take the shape of a star network, which makes it a popular solution in literature. It should be noted that other network topologies such as point-to-point, mesh, and tree also exist and can commonly be seen at different layers of an IoT system, as the choice of topology is oftentimes a byproduct of the reliability and throughput requirements of the network as well as the characteristics of its elements [1].

The majority of communications in a Smart City environment are wireless, with the concrete choice of physical layer heavily depending on its use case. The available power, the intended transmission range, and the necessary data rate are the key aspects that must be considered. Constrained devices, in particular, require efficient networking protocols to operate effectively.

Energy-efficient solutions like 6LoWPAN, Bluetooth / Bluetooth Low Energy (BLE), Z-Wave, and Zigbee are popular for short-range communications. At the same time, on the long-range spectrum, low-power wide-area networks (LPWANs) like LoRaWAN are predominant. Wi-Fi and Cellular technologies (4/5G) are standard solutions where high data rates are necessary over medium to long distances, network coverage is available, and power constraints are more relaxed [1].

It should be noted that wired networks are also possible options; however, the need for cables and all associated infrastructure often impede their usage for IoT applications and, as such, are not in the scope of discussion.

Another integral part of IoT communications is the choice of the messaging protocol. Unlike the Web, which uses HTTP as its default messaging protocol, the IoT cannot rely on a single protocol to meet all use cases, due to the heterogeneous nature of IoT devices and applications [46]. As such, several protocols have been developed for different types of requirements, such as applications requiring fast and reliable business transactions (AMQP), data collection in constrained networks (MQTT, CoAP), web applications with REST APIs (CoAP, HTTP), to name a few. This

means that, much like when choosing a network protocol, the messaging protocol must be selected per the application requirements and hardware constraints [46].

Interoperability at this level can be achieved by limiting the number of protocols in use simultaneously by the same network and by introducing gateways – nodes capable of translating between different protocols. The latter is a prevalent approach supported by various data models and IoT ecosystems, such as the ones presented in the previous sections. Some works also propose systems that seek to support various protocols on constrained devices directly. Oliveira et al. [47] propose such a framework. Using a Service Oriented Architecture (SOA), the authors create a runtime for constrained devices that is capable of, among other things, supporting a variety of different protocols through the usage of dedicated and interchangeable software modules.

3.5 Summary & Discussion

This Chapter discusses various works, technologies, and ideas that have either directly influenced or present alternative solutions and approaches to the problems this dissertation seeks to tackle.

As stated in section 3.1, the W3C Web of Things standard was chosen as the technological foundation and base vocabulary for this dissertation work. First, as a W3C Recommendation, the WoT has garnered interest in both industrial and academic fields as presented in the survey in [8]. Moreover, the technological building blocks provided by the standard, namely, the *Thing Description* and the TM, offer a rich base vocabulary for describing services, APIs, and characteristics of *Things* in an IoT infrastructure.

Another benefit of integrating the W3C WoT into the framework is adhering to the semantic web paradigm. This paradigm represents a conceptual evolution of the traditional World Wide Web, primarily driven by standardization efforts from the W3C and focused on enabling machines to understand the meaning (i.e., semantics) of data through a graph-based representation, where an Internationalized Resource Identifier (IRI) identifies each node. These nodes can then be connected by establishing associations between humans and machines that can be parsed and followed to discover how the different concepts expressed in the data are related.

The WoT adheres to the Semantic Web paradigm via the Resource Description Framework (RDF) [17], [48], a W3C Recommendation for representing linked data graphs. Additionally, the JSON-LD [19] provides a human-friendly serialization format for RDF data based on JavaScript Object Notation (JSON). Consequently, the TM and *Thing Descriptions* from WoT are essentially RDF graphs represented as JSON-LD documents that humans and machines can easily read and manipulate.

Compared to the WoT, NGSI-LD focuses more on modeling the properties and characteristics of *entities* in an IoT system and does not provide a standard format for describing the communication APIs between *Things* in the network. This is a desirable feature for adaptive IoT infrastructures [5], which the WoT offers as the *Thing Description*. Additionally, [10], [43], and [11] have all created extensions and adaptations of the FIWARE data models to model their particular

systems. While flexible, this may challenge larger-scale systems where arbitrary deviations from the established models may provoke fragmentation.

While both FIWARE and oneM2M provide an extensive selection of software services to work with their respective models, the WoT, being more recent, does not yet have such an ecosystem surrounding it. Efforts are being made in this direction, primarily by the *Eclipse Thing Web* project [49], which provides the Node-WoT [39] library to facilitate application development in compliance with the Web of Things standards.

Even so, WoT technological building blocks can be successfully used for real-world projects as shown in [14], [38], and [8]. Similarly, the solution created in this work, denominated as the *CityLink* framework, leverages the W3C WoT to provide detailed and semantically rich models of the physical system without locking in the network architecture into a particular ecosystem of services.

Some of the works tackle IoT interoperability at the edge node layer, working with hardware capable of virtualization and other sophisticated software features [11], [14], [43], [44]. This is not the primary development direction for the *CityLink* framework. While complementary services were implemented at the edge node level, the proposed solution aims to encompass and model all edge participants in an IoT system, including first-class support for constrained end nodes, in charge of interfacing with the physical world.

This aspect is more in line with the works of [13], [38], and [47]. In [38], the authors affirm the feasibility of using the WoT *Thing Descriptions* and the scripting API to model and connect real sensing hardware to cloud services. *CityLink* aims to go further, taking advantage of TMs to streamline the creation process of *Thing Descriptions* and establish (linked data) relations between the various elements of the system, like the one supported by NGSI-LD.

Going further, [13] brings the WoT to constrained end-devices. While this may fill the gap left by previous works, their approach does not consider the orchestration aspect necessary for larger-scale deployments, nor does it provide an extensible data model essential for dynamic systems. Adopting the Arduino tooling leads to an opt-in into the respective ecosystem, limiting hardware and software choices. Similarly, the approach taken by [47] provides adaptability and interoperability at the communications level, with devices being able to receive over-the-air updates and changing messaging protocols dynamically. The authors, however, do not propose an overarching data model for the IoT system, nor take into account semantic interoperability between applications, with their framework being designed around *ad-hoc* JSON configuration files.

An apparent gap in the literature can be identified. Existing works targeting interoperability at the edge tend to focus on more capable edge nodes or constrained end nodes, without treating both as integral parts of a unified system architecture. This is precisely the gap that the *CityLink* framework seeks to address. By providing a semantically expressive and hardware-agnostic modeling approach, *CityLink* enables representation, management, and orchestration of large-scale IoT networks, focusing on Smart City scenarios. Nonetheless, the framework remains flexible enough to be adapted for other domains requiring distributed sensing and actuation.

Semantic interoperability is achieved by adopting the W3C *Web of Things* standard, whose foundational constructs enable structured and machine-readable representations of IoT resources. On the other hand, hardware interoperability is facilitated through newly proposed software components and a system architecture designed with modularity and extensibility in mind. In contrast to other standards discussed, such as NGSI-LD or oneM2M, the WoT presents a lower entry barrier, making *CityLink* more accessible to developers while maintaining expressive power and scalability.

As a final note, it is worth emphasizing that although the W3C *Web of Things* remains an emerging standard, it is already influencing the direction of more mature frameworks. For instance, the ongoing development of oneM2M Release 4 (with Release 5 in planning) introduces features like WoT Interworking and Semantic Reasoning with Ontology Mapping, explicitly aiming to reconcile oneM2M's resource-centric model with the semantic Web. These developments open the door to future integration between *CityLink* and the broader oneM2M ecosystem, as well as with works such as [44], which focus on virtualized service environments at the network edge.

In summary, this dissertation positions *CityLink* as a lightweight yet powerful approach to bridging semantic, hardware, and system-level gaps in distributed IoT deployments, aligning with current trends while anticipating future interoperability requirements in scalable, real-world infrastructures.

Chapter 4

CityLink Framework

The *CityLink* framework leverages the W3C Web of Things to provide a comprehensive solution for managing end nodes in large-scale IoT deployments. The proposed solution targets the development of an adaptable edge layer infrastructure for smart cities, focusing on the dynamic reconfiguration of end nodes and the high flexibility in developing and deploying new applications. Compute in urban environments is typically organized as illustrated in Figure 4.1, where edge layer servers (edge nodes) act as managers and gateway devices that connect the sensing/actuating infrastructure (end node) to various cloud services and applications. In this context, interoperability is supported at the hardware and communications level by implementing specialized software components and at the data/semantic level by the W3C Web of Things Standard [31].

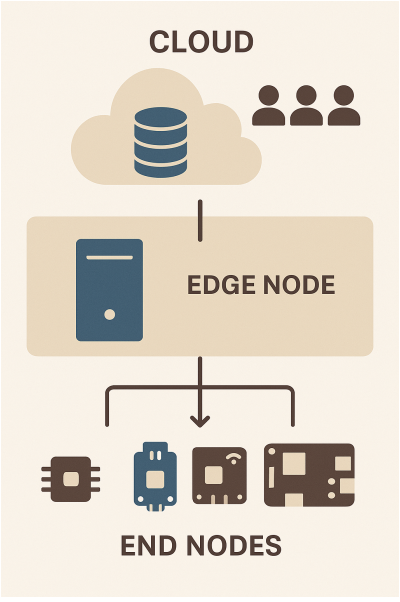


Figure 4.1: Cloud-Edge computing paradigm.

It is worth mentioning that to achieve end node interoperability, the proposed framework remains agnostic to the underlying communication protocols and network infrastructure. Therefore,

this proposal assumes that the necessary infrastructure is in place, allowing the framework's components to communicate seamlessly. Such network and protocol agnosticism facilitates interoperability in urban environments, where highly heterogeneous network configurations are commonly found. Based on this, the following assumptions are made and must be understood as part of the framework's definition:

1. The end nodes can connect to edge nodes on their respective networks.

2. *Embedded Core* and *Edge Connector* instances share the same messaging protocol, data serialization formats, and common APIs for communication.

Network compatibility must be enforced at the hardware level, as network technology is not expressed as part of the WoT vocabulary. Conversely, messaging and data standardization can be managed by the WoT since protocols, communications endpoints, data payloads, and serialization formats are inherently expressed.

4.1 Framework Overview

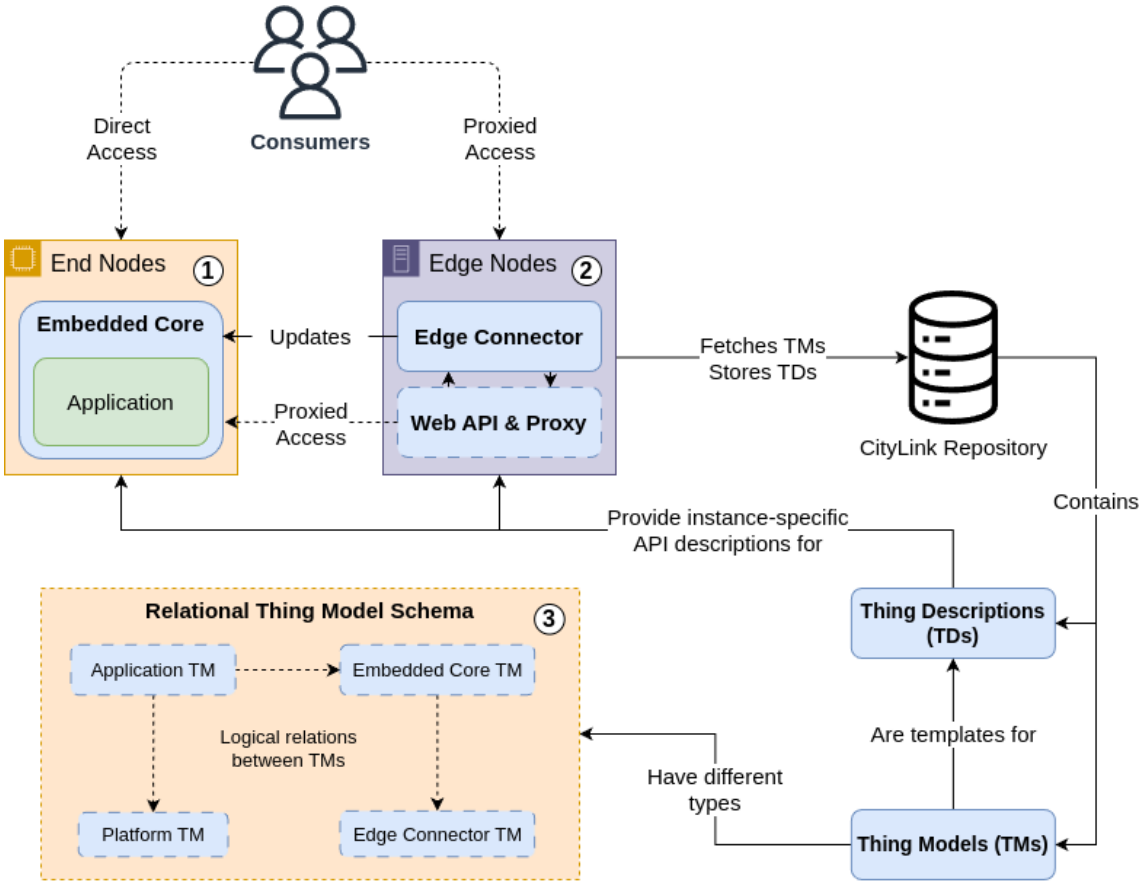


Figure 4.2: Model-based adaptable framework for dynamic configuration of sensor platforms.

Figure 4.2 represents the operational components workflow for the *CityLink* framework. Following the numbering in the figure, the main elements of the proposed framework are described as follows:

① **End Nodes:** These are typically embedded platforms that execute sensing or actuating functions. End nodes run the *Embedded Core* firmware, a key enabler of end node adaptability and hardware interoperability. These devices comprise the bulk of the IoT system and are responsible for collecting data from the environment or performing actions based on the information received. The modern IoT ecosystem comprises various end nodes communicating over physical and logical networks, such as Wi-Fi, LoRaWAN, Zigbee, etc. Supporting low-power and low-cost devices at this level is essential for the system's scalability and economic viability of large deployments.

② **Edge Nodes:** More powerful platforms can be deployed strategically to manage the end nodes in their vicinity, acting as gateways to the fog, cloud, or other edge devices. Edge nodes typically support full-fledged operating systems with virtualization capabilities capable of running multiple applications and services. Within the context of the CityLink framework, these devices are characterized by a key software component, the *CityLink Edge Connector* service. This service allows edge nodes to manage the life cycle of end nodes, including their automatic registration and adaptation of new and existing end nodes. Furthermore, edge nodes can offer proxy services for end nodes, replicating their state and providing a more responsive interface for consumers, bypassing the need for direct communication with the constrained devices.

③ **Relational Thing Model Schema:** This abstract component defines the relations between the different elements of the proposed framework. Regardless of the underlying hardware, communication protocols, or implementation details of the components above, the framework provides a set of rules and guidelines that govern how these elements interact. In practice, this component dictates the contents of various TMs that must exist and how they are related. The documents can then be stored in an online repository, linked data stores, or other database systems, as long as they are accessible to edge nodes.

In the context of this work, a *consumer* is any entity compatible with the WoT semantic infrastructure that consumes data or controls the services provided by the end nodes. Communication between end nodes, edge nodes, and consumers always adheres to the WoT *property-action-event* paradigm.

In an urban environment, web services could consume distinct *Property Affordances* to deliver environmental data to the city's inhabitants through a web interface. Conversely, an administration platform could be the leading consumer of *Event Affordances*, enabling city officials to gather diagnostics and alerts for maintenance and supervision. Machine learning algorithms could also be employed to process the collected data in the cloud and leverage the *Action Affordances* of various end nodes for actuation or even dynamic configuration of the sensing functionality across the

city, thereby adapting the system to urban dynamics. Additionally, consumers may have different access levels and rely on a subset of the *Interaction Affordances* provided by a single end node.

The *Relational TM Schema* is the main contribution of this work, as it establishes logical connections between different system components, along with a set of software components that enable automated registration and configuration of end nodes. Additionally, the proposed software components facilitate the translation of the APIs derived from the data models into code, allowing for a streamlined application development process, focusing on the logic while abstracting the communication interfaces. The following sections are dedicated to establishing the fundamentals of the functional operation based on the proposed set of abstractions.

4.2 End Node Life Cycle

With *CityLink*, the life cycle of end nodes is divided into four main phases, *Initial Configuration*, *Registration*, *Normal Operation*, and *Adaptation*. These phases are represented in Figure 4.3, which illustrates how each life cycle phase is interconnected, providing a high-level overview of an end node's functional operation.

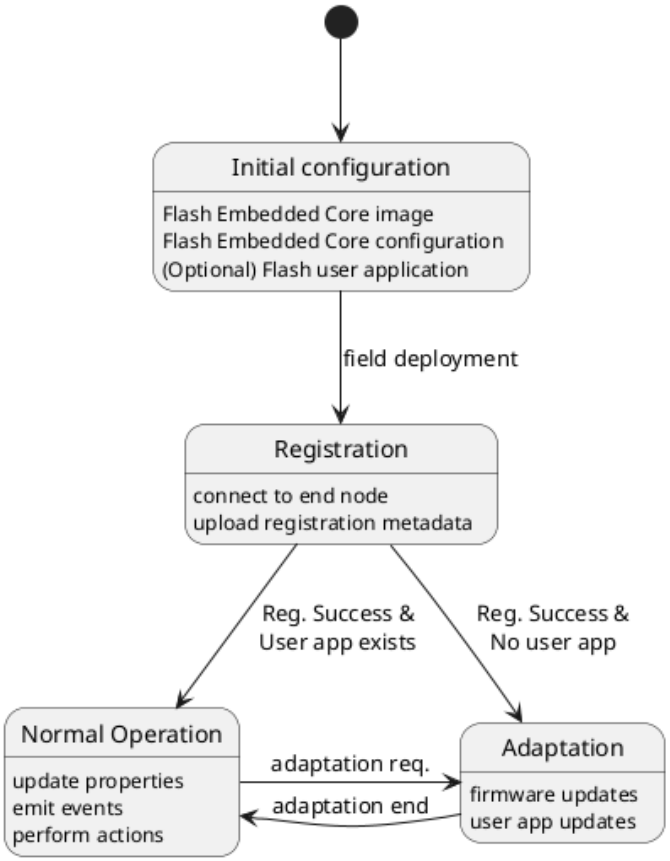


Figure 4.3: CityLink End Node life cycle

4.2.1 Initial Configuration

The first step in the life cycle of an end node is its *initial configuration*, which occurs before deployment. This process involves configuring the end node with the *Embedded Core* firmware, which provides a runtime that supports the essential features of the proposed framework.

An end node may also be pre-configured with an *application*, a piece of software developed using the *Embedded Core* API that encapsulates the business logic to be executed locally on the device. This *application* defines how the end node behaves in its operational context once registered with the broader system. For instance, the *application* might continuously monitor and report temperature readings, or control the brightness of a streetlight based on environmental and scheduling data.

The *Embedded Core* runtime simplifies the development of such applications by exposing a consistent API and runtime environment that aligns with the system's underlying data model and communication standards. This ensures that applications remain interoperable across heterogeneous devices and can be dynamically reconfigured or extended over time, provided the hardware remains compatible.

End nodes must also be configured with a URL pointing to their associated TM in this application deployment phase. This model defines the device's expected capabilities and *Interaction Affordances* in a machine-readable format, serving as a contract between the end node and the broader IoT framework.

4.2.2 Registration Procedure

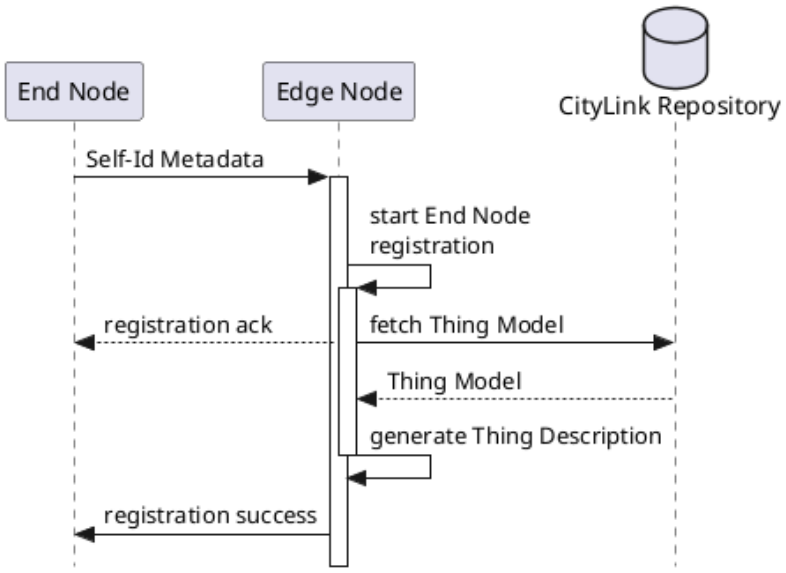


Figure 4.4: Sequence diagram for End Node registration procedure

The *registration procedure* is the process by which a new end node connects to and integrates with the city's IoT infrastructure. When an end node first boots, it attempts to connect to an

edge node. It then sends its self-identification metadata (the TM URL, allowing the edge node to identify the end node, including its hardware characteristics, capabilities, and the application it is running. Depending on the application and the edge node's requirements, end nodes may provide additional metadata as part of the registration payload. This should be avoided, however, as offloading information to TMs is preferred. Doing so promotes reusability and modularity and reduces the amount of data that needs to be transferred between the end and edge nodes. End nodes store only a reference to their model, not the model itself, as semantically rich TMs, and generated *Thing Descriptions*, can get extensive and be too large and take up precious resources if stored directly on constrained devices.

Once the edge node fetches and validates the TM that the end node indicates, a comprehensive *Thing Description* can be generated by enriching the TM with instance-specific metadata extracted from the self-identification metadata provided by the end node and the edge node's configuration.

All of the information fetched and generated during the registration procedure is then stored/cached (in-memory, on disk, in a remote database, etc.) by the edge node to be available for future requests and to be served to consumers upon request. Consumers can use *Thing Descriptions* to discover the API endpoints and interact with the services the registered end nodes offer.

Successfully registered end nodes are then assigned a *unique identifier* by the edge node, which tracks the end node's state and interactions within the network. This identifier is essential for the edge node to manage the end node's life cycle, including its adaptation and normal operation phases.

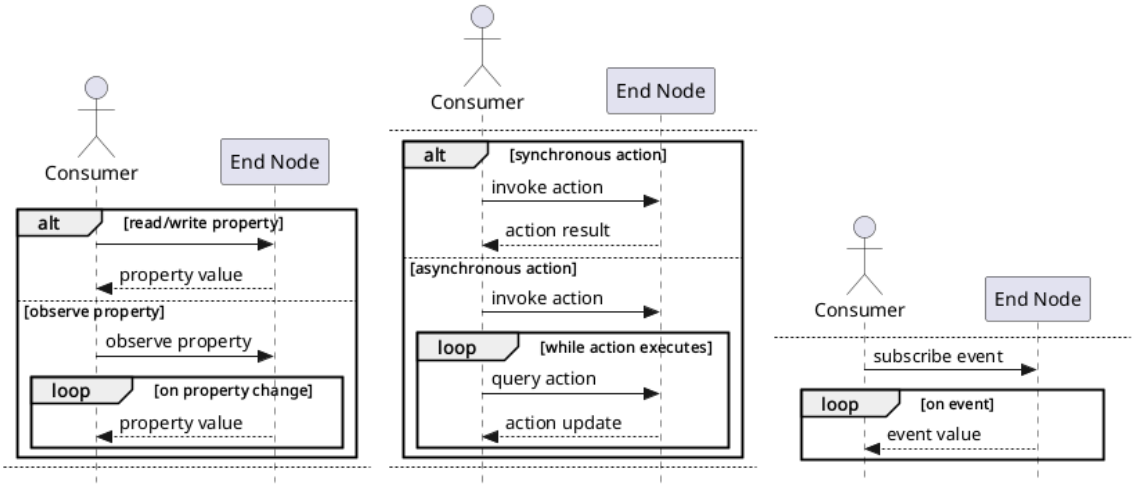
Once this process is completed, the end node is considered registered. It may either enter the *Normal Operation* phase (4.2.3) or immediately undergo an adaptation procedure (4.2.4) to update its application. As shown in Figure 4.3, the choice of which phase to enter next is determined by the presence of an application when the end node is initially deployed.

4.2.3 Normal Operation

A registered node with executing application code is in the *Normal Operation* phase of its life cycle (Figure 4.3). In this phase, the end node operates as a regular *Thing* in the WoT ecosystem, providing its services through the APIs defined by its *Thing Description* and *Thing Model*.

Application code may execute periodic tasks, such as sampling sensors, executing actions, or emitting events, depending on the available hardware and the application logic. As mentioned, interaction with consumers is always done through WoT *Interaction Affordances*. Properties may represent configurable parameters that consumers can read or write (Figure 4.5a read/write property block), or sensor data that is periodically sampled. In such cases, it may be beneficial for consumers to *observe* such properties, registering them for asynchronous value updates (Figure 4.5a observe property block).

Conversely, actions can be used for consumers to trigger specific behaviors in the end node, whether a simple command to turn on a light or a more complex operation requiring the end node to perform a sequence of operations. Figure 4.5b illustrates this interaction. On the *synchronous action* block, the consumer sends a request to the end node, which then executes the action and



(a) Property Interactions (b) Action Interactions (c) Event Interactions

Figure 4.5: Normal Operation sequence diagram

returns a response. For asynchronous actions, an initial response may (or may not) be returned immediately, leaving the consumer to query the end node for the action's status. (Figure 4.5b asynchronous action block).

Finally, events may help signal abnormal states, publishing log messages, or pushing notifications to consumers (Figure 4.5c event block). This interaction follows the same logic as *observable* properties. It is analogous to the publish-subscribe pattern, which is widely used in IoT systems to decouple producers and consumers of data.

4.2.4 Adaptation Procedure

Either during *Normal Operation* or as a complement to the *Registration* procedure, a *CityLink* end node may need to be adapted.

In normal operation, a consumer (admitting proper authorization) may request an edge node to initiate an adaptation procedure for one of its registered end nodes. This request reassigns the end node's TM from which new application features and resources can be resolved, with the edge node automatically managing the update process. In a Smart City context, AI-powered systems automatically initiate end node adaptations to adapt the urban IoT infrastructure to new variables or dynamic environmental changes.

Conversely, the end node can also initiate the adaptation procedure, signaling the need for an update to its application or configuration. Generally, this adaptation is not coupled with a TM change, acting only as a means for end nodes to receive missing application resources or configuration updates immediately upon registration.

Figure 4.6 illustrates the sequence of events as the adaptation procedure unfolds. In the top part of the figure, under the *End Node Adaptation Mode Activation* label, both alternatives are shown, where either a consumer triggers the adaptation procedure, sending the updated TM to the edge node, or the end node itself signals the need for an update.

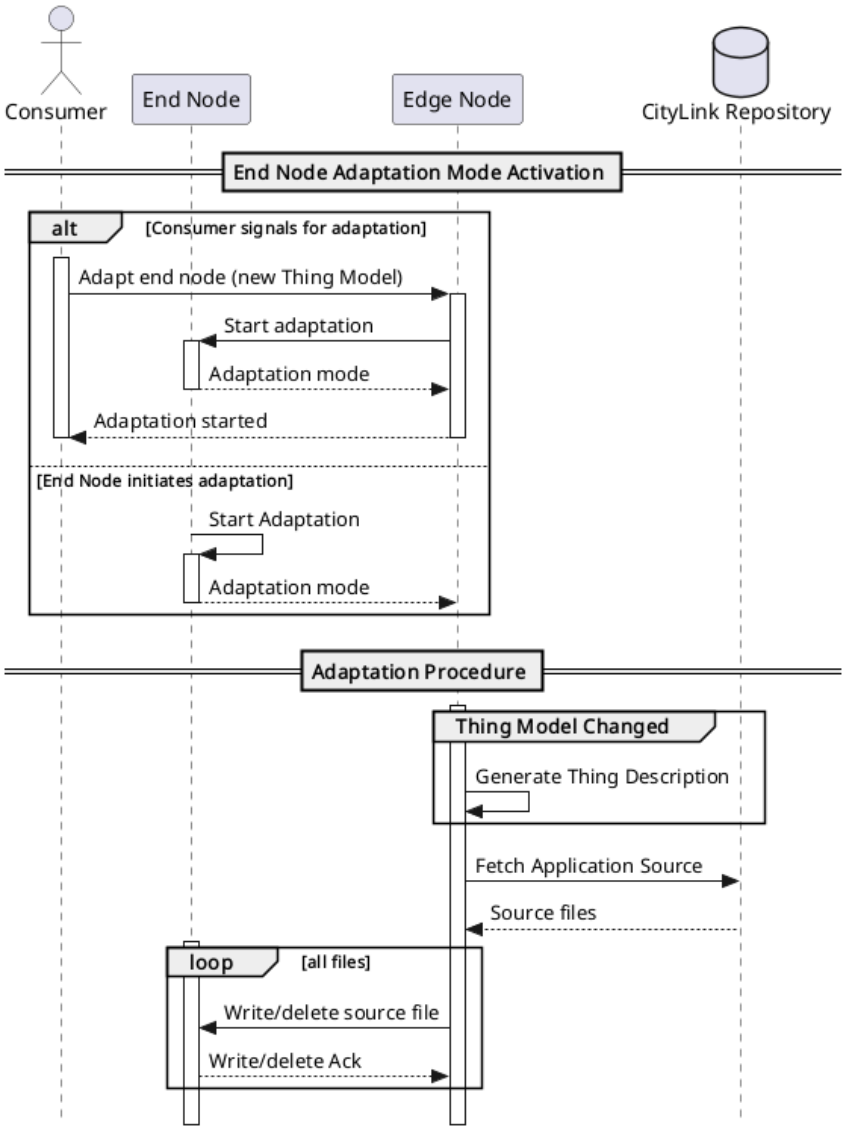


Figure 4.6: Sequence diagram for end node adaptation.

Once the end node enters the adaptation mode (rebooting or resetting into a safe boot state), the edge node begins the procedure. Inside the *Thing Model Changed* block in Figure 4.6, the edge node fetches the new TM and the application code from the specified URLs and generates a new *Thing Description* for the end node.

The actual update of the end node's application is performed according to the capabilities of the particular *Embedded Core* version running on the end node. For this section, the *Loop* block represents an iterative process of uploading individual source files (application code) to the end node. This process can be easily implemented in scripting environments that support virtual file systems in constrained devices. This is the approach described in the Chapter 6 for implementing the proof-of-concept *Embedded Core* runtime in MicroPython. Other implementations, however, would behave differently. With compiled languages, for example, the application code would

be compiled into a binary image that is then uploaded to the end node in one go, replacing the previous application code. If dynamic linking is supported, the edge node may only upload the new or changed files, leaving the rest of the application code intact.

These details are implementation-specific and are not part of the framework's definition, as they do not affect the interoperability of the end nodes with the edge nodes or consumers.

4.3 Embedded Core

The *Embedded Core* is the software component that enables the operation of end nodes within the *CityLink* framework. It provides the necessary runtime environment for end nodes to execute their applications and interact with *CityLink* edge nodes and consumers. The main goal for this piece of software is to ensure that end nodes can be easily registered, adapted, and interoperable with the rest of the framework, regardless of the underlying hardware or communication protocols. This means that the *Embedded Core* must be designed to be flexible and adaptable, allowing for different implementations and protocols while maintaining a consistent API for application development. Figure 4.7 provides a more detailed view into the software architecture expected for *CityLink* end nodes previously presented in block ① of Figure 4.2.

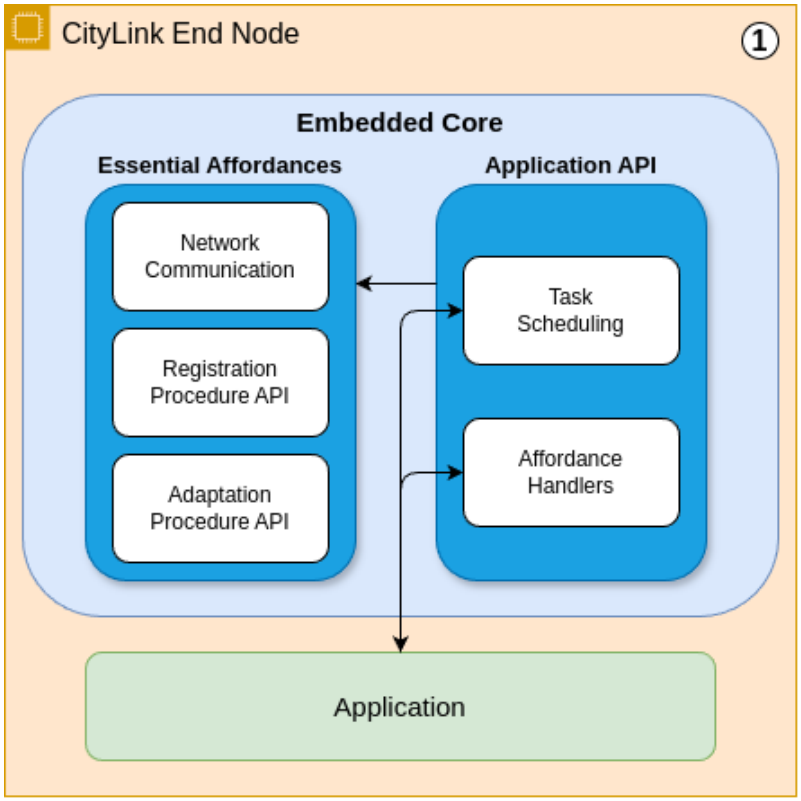


Figure 4.7: Detailed CityLink End Node software architecture

Embedded Core implementations should conceptually be divided into two separate layers:

- essential *Interaction Affordances*.
- application API.

The *essential Interaction Affordances* supported by the runtime must encompass all the network communication features and any additional functionality required to support the registration and adaptation of end nodes. This ensures that end nodes can, at minimum, connect to edge nodes, carry out registration, and undergo adaptation, regardless of the application code running on them. The base feature set can then be extended with an *Application API* that provides a higher-level interface for application development, allowing developers to create applications that can interact with the end node's hardware and services.

This separation of concerns highlights the critical functionality that must always be ensured by *Embedded Core* implementations. Should the application code not be present or crash during execution, end nodes within the *CityLink* framework should still be able to connect to edge nodes and receive updates, ensuring that remote management and maintenance are always possible.

Striving to maximize interoperability, it is expected that multiple versions and implementations of *Embedded Core* runtimes may exist, targeting different platforms and communication protocols. These can also be implemented with various languages and technologies such as MicroPython [50] and bare metal C/C++ or over an RTOS such as Zephyr [51]. This allowed diversity highlights the framework's interoperability, as platforms may follow different architectures and require specific firmware, while still being compatible with the overall data models.

In the face of this diversity, *Embedded Core* implementations can be organized into families based on their implementation language and target communication protocol (e.g., MicroPython and MQTT, Zephyr and CoAP, Rust and HTTP, etc.). Runtimes within the same implementation group should be derived from a base, protocol-agnostic TM that outlines the general API for application development. Applications developed for MicroPython, for example, should be compatible with any MicroPython-based *Embedded Core*, regardless of the underlying communication protocol, with the protocol-specific bindings being abstracted away from the application code by the high-level application API. The *Affordance Handlers* block in Figure 4.7 illustrates this concept, where the application API provides a set of handlers for each *Interaction Affordance* type (properties, actions, and events) that can be used by the application the translate the TM into code using a straightforward approach. A concrete example can be found later in Section 6.2.1.

Table 4.1 summarizes the relationship between the communication protocols and implementation families of *Embedded Core* runtimes and how they affect both the registration and adaptation procedures and the application development process.

The first row of the table highlights that the communication protocol used by the *Embedded Core* implementation determines how end nodes interact with edge nodes and consumers, meaning that the *Interaction Affordances* (properties, actions, and events) are directly influenced by the protocol used. For example, producer-consumer protocols such as MQTT might need to map

Table 4.1: *Embedded Core* communication protocols, implementation families, and their effect on the end node registration, adaptation, and application development.

	Registration Protocol	Adaptation Protocol	Application Development
Communication Protocol	Interaction Affordances	Interaction Affordances	Mostly Agnostic
Implementation Family	Agnostic	Data payloads OTA Update Algorithm OTA Update Granularity	API Performance HW Drivers

request-response interactions to pairs of *Interaction Affordances* where the request is an action and the response is a property or event. Conversely, client-server protocols such as HTTP do not require such mapping, but may need additional support for asynchronous interactions such as observable properties and event subscriptions. This aspect is primarily reflected in the algorithms that register and adapt end nodes. For application development, Table 4.1 indicates that application development is **mostly** agnostic. While the application code is portable across different *Embedded Core* implementations in the same family, its outward-facing API may differ slightly for the same reasons as the registration and adaptation procedures. In practical terms, while end node applications might be protocol-agnostic, consumer applications may suffer protocol-specific adjustments as the communication endpoints and interaction models do not directly translate between protocols.

Taking a look at the second row of Table 4.1, it is possible to conclude that, while the registration procedure is agnostic to the end-node implementation characteristics, the adaptation procedure as well as the application code are not. The architecture of the *Embedded Core* not only determines how and what kind of OTA updates are supported (as mentioned in Section 4.2.4), but also how the application code is developed and executed.

Further evidence of this aspect is presented in the remaining block in the application API layer in Figure 4.7: the *Task Scheduling* module. As its name suggests, the module provides functionality that applications can use to schedule tasks to be executed periodically, once, at a specific time, or after a delay. While this is not strictly part of the *CityLink* information model, it is a common requirement in embedded applications. It is highlighted as a key technical requirement for any practical, non-trivial end node deployment in an IoT system. Depending on the underlying hardware and programming language/technology, task scheduling may be implemented via asynchronous programming, cooperative multitasking, or preemptive scheduling.

The following sections (4.4 and 4.5) provided further details into the how the *Embedded Core* interacts with the rest of the framework, and how the *Relational TM Schema* is used to define the relationships between the and enforce the API contracts between the various elements.

4.4 Edge Connector

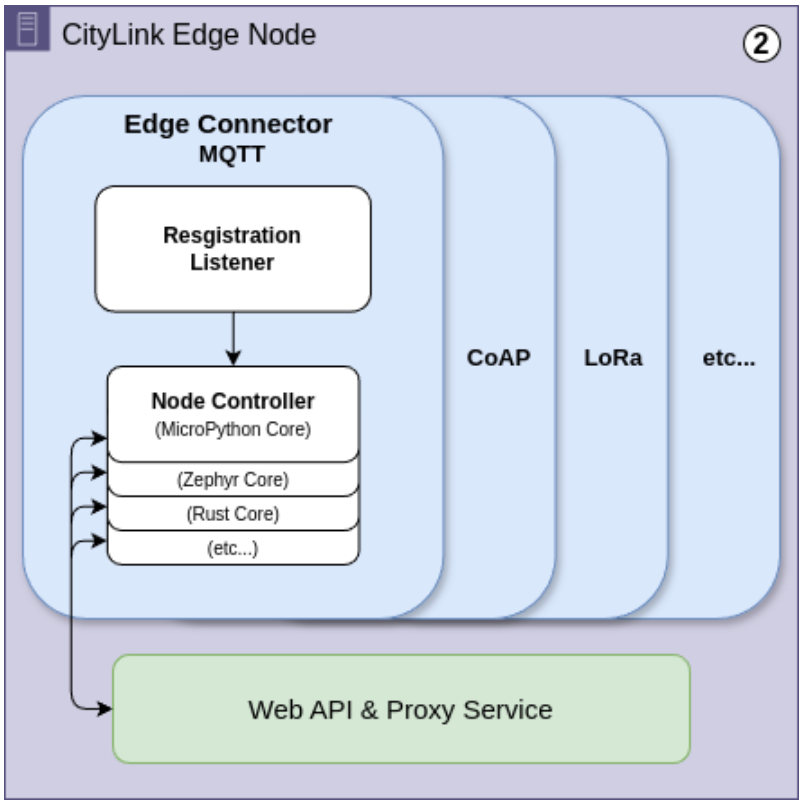


Figure 4.8: Detailed CityLink Edge Node software architecture

A matching counterpart to the *Embedded Core* is the *Edge Connector* service, which is a software component that runs on *CityLink* edge nodes. In IoT systems, edge servers usually serve as gateways or intermediaries between end nodes and consumers, providing a layer of abstraction and management for the end nodes they oversee. This is equally true for IoT deployments that adhere to the proposed framework. As with the *Embedded Core*, it is expected that multiple implementations of the *Edge Connector* service may exist and target different scenarios and network architectures. In light of this, this section provides the general architecture and expected behavior for this software component, offering some general implementation guidelines derived from the envisioned functional operation of the framework.

Figure 4.8 illustrates the software architecture of a *CityLink* edge node, highlighting the plurality of *Edge Connector* instances that can run on a single edge node, catering to different communication protocols (given adequate hardware support).

As the *Edge Connector* instances must adequately match with the end node they manage, the conceptualized modular architecture for this component comprises two main elements:

- **Registration Listener:** This component is responsible for listening for incoming registration requests from end nodes and initiating the registration procedure. Its functional operation is described in Section 4.2.2, where it receives the end node's self-identification metadata, fetches its TM, and generates a *Thing Description* for the end node.
- **End Node Controller:** To handle end node adaptation and to serve as a dedicated manager module for registered end nodes, the *[End] Node Controller* is a software component that directly matches the *Embedded Core* implementation of the end node it manages. It follows that, as part of registration, a compatible controller must be selected and instantiated for each end node. Figure 4.9 provides some additional insights into this.

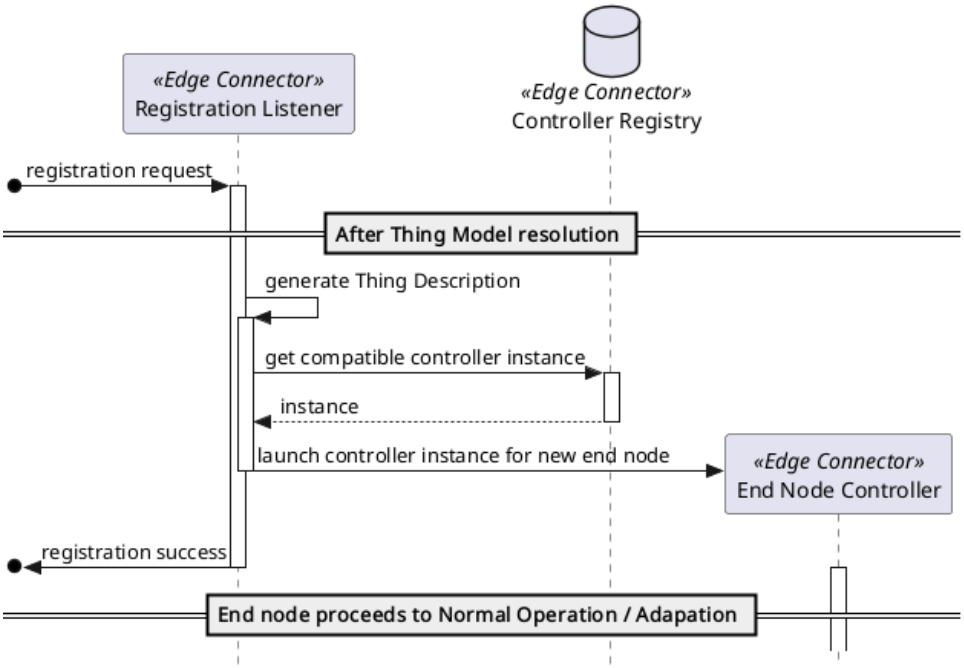


Figure 4.9: Detailed sequence diagram of the final steps of the end node registration procedure

Figure 4.9 illustrates how the *Registration Listener* creates a new *End Node Controller* instance for each end node upon registration. This occurs after the initial sequence of events shown in Figure 4.4, but before the registration process is acknowledged as successful.

To support this mechanism, the *Edge Connector* includes a complementary component: the *End Node Controller Registry*. Represented with a database icon in the figure, this auxiliary structure emphasizes a key architectural goal of the *CityLink Edge Connector*: the ability to manage multiple controller implementations—each tailored to a specific class of end node hardware—under a unified API.

The Controller Registry is a central part of this design, enabling the edge node to dynamically choose the correct controller for a given end node based on its associated TM. While the specific implementation details of this mechanism are beyond the scope of this chapter, a practical example is presented in Chapter 6 as part of the evaluation and demonstration of the system.

4.4.1 Web API & Proxy service

In green, Figure 4.8 presents two additional features that, while not being essential, are expected to be provided by *CityLink Edge Nodes*: A *Web API* and a *Proxy Service*.

The *CityLink* framework is designed to be flexible and adaptable, allowing for various deployment scenarios and hardware configurations. Depending on the concrete deployment scenario and the underlying hardware characteristics of the different elements of the IoT system, it may be advantageous for edge nodes to proxy the *Interaction Affordances* of the end nodes they manage.

One example use case is for non-IP networks. Despite the advantages of the W3C WoT paradigm, its information model is poorly equipped with tools to describe *Things* that do not employ traditional Web technologies. This makes it challenging to create TM when the communication protocol cannot be easily expressed or no protocol-specific bindings exist.

In *proxy mode*, the edge node acts as an intermediary between the end node and consumers, caching incoming properties and events and forwarding actions. The edge node can be a gateway or translation unit between different protocols.

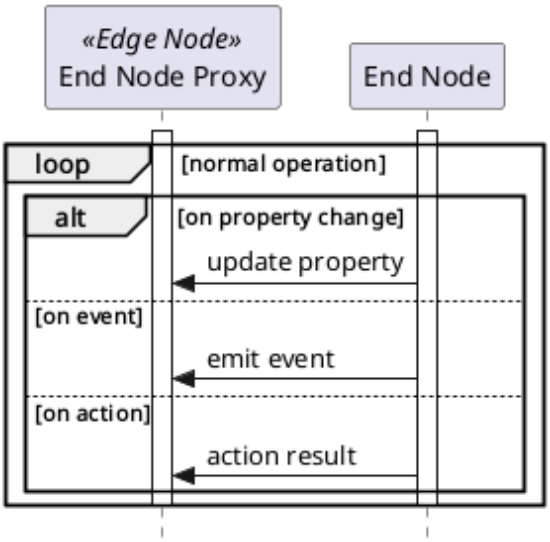


Figure 4.10: Sequence diagram showing proxy being continuously updated by the end node

Figure 4.10 illustrates how the proxy service works in practice. During normal operation, a registered end node continuously updates the edge node with the latest values for its properties, events, and any results from invoked actions. In publish-subscribe protocols, such as MQTT, this is easily achieved by having the end node controller on the edge node subscribe to all *Interaction Affordances* of the end node it manages and forwarding them to the proxy service. For other protocols, the edge node controller may need to implement a polling mechanism to retrieve the latest values from the end node.

The proxy service then serves as a cache for the end node's state, allowing consumers to interact with the end node without communicating directly. This is particularly useful for constrained

devices that may not have the resources to handle multiple simultaneous requests or that may be located in remote areas with limited connectivity. Some *Edge Connector* implementations may even be necessarily implemented as a proxy service, particularly when dealing with non-IP networks and/or low bandwidth communication protocols such as LPWANs (e.g., LoRaWAN). Finally, the proxy service can also provide additional features, such as caching, rate limiting, and security, to ensure that the end node's resources are used efficiently and securely.

The proxy service is transparent from a consumer's perspective, providing the same API endpoints as the end node's *Thing Description*. This means that consumers can interact with the end node as if communicating directly, without knowing that the edge node is acting as a proxy. This transparency is achieved by having the edge node's *Edge Connector* service automatically generate and update the *Thing Description* for the end node, including the proxy service's API endpoints.

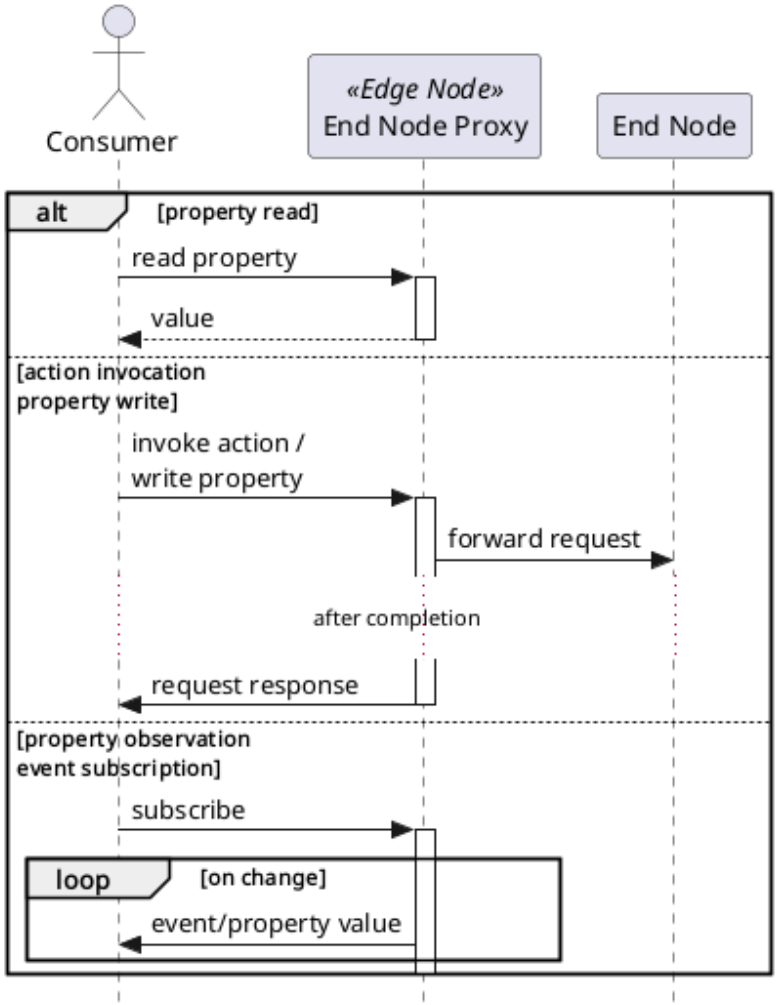


Figure 4.11: Sequence diagram showing proxy service interactions with consumers

Figure 4.11 illustrates how the proxy service interacts with consumers, with read requests being served directly from the proxy cache, while write requests are forwarded to the end node.

Proxy operation is a well-defined part of the W3C WoT specification [30], is transparent for consumers and end nodes, and is a common practice in IoT systems where constrained devices are involved. The *CityLink* framework leverages this concept to provide a flexible and adaptable solution for managing end nodes in urban IoT deployments.

Complementing the *Proxy Service*, edge nodes may also provide a *Web API* that provides consumers with a RESTful interface to discover and manage *Thing Descriptions* of registered end nodes, per the W3C WoT Discovery specification [32]. If a proxy service is available, the *Web API* can also provide access to its API endpoints, providing a unified interface where consumers can discover and interact with the end nodes they are interested in.

The WoT Discovery specification defines a set of APIs that allow consumers to discover *Thing Descriptions*, perform semantic searches, and retrieve metadata about the available *Things* in the network as well as perform CRUDL (Create, Read, Update, Delete, and List) operations on them. For *CityLink* edge nodes, however, full compliance with the WoT Things API should be implemented with care, as allowing for the creation and deletion of *Thing Descriptions* by consumers may lead to inconsistencies in the system, bypassing the registration and adaptation procedures defined in Section 4.2 and potentially leading to inconsistencies with the *Relational TM Schema*.

A recommended approach is to fully support both the *Search* and *Events* APIs as defined in the WoT Discovery specification, allowing consumers to discover *Thing Descriptions* and receive notifications for newly registered end nodes. CRUDL operations other than READ and LIST should be highly restricted, allowing only authorized management applications or system administrators to perform such operations. Ideally, the *Web API* should also be extended with *CityLink*-specific endpoints to allow consumers to issue adaptation requests for registered end nodes, as described in Section 4.2.4 and browse known TMs, circumventing the need for direct CRUDL operations on *Thing Descriptions* and ensuring consistency of the data model and the physical devices.

4.5 Relational Thing Model Schema

Briefly addressed in the previous sections, the *Relational TM Schema*, represented in Figure 4.2 block ③ and, in greater detail, in Figure 4.12. This component is the glue that binds the various components of the *CityLink* framework together, providing a standard data model and API contract for end nodes, edge nodes, and consumers. While the *Thing Model* is optional for compliance with the W3C WoT, it plays a central role within *CityLink*. This model dictates the features the end node implements and provides the template used to instantiate valid *Thing Descriptions* during registration or after adaptation.

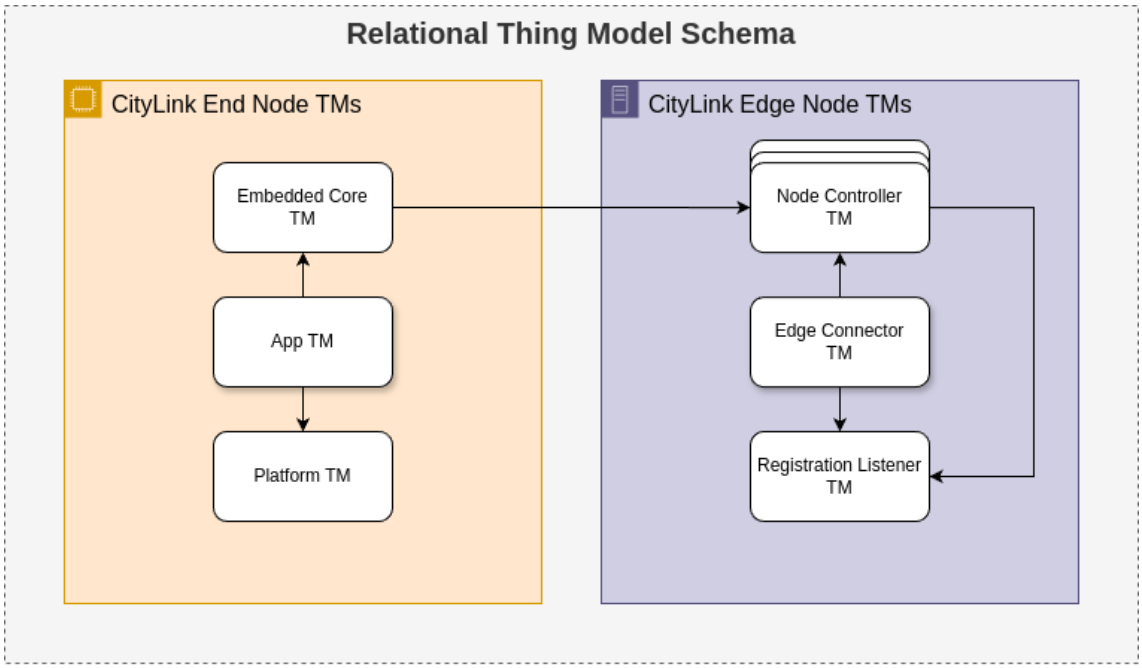


Figure 4.12: Relational TM Schema Overview

1. **End Node TMs**:

   For each end node, three aspects must be modeled: The *Embedded Core* firmware running on the end node, the end node's hardware characteristics, and the end node's application logic. For this purpose, the *CityLink Relational TM Schema* defines three distinct TM types that are used to describe these aspects:

   - **Embedded Core TM:** Models the API (as *Interaction Affordances*) of the *Embedded Core* firmware running on the end node.
   - **Platform TM:** This model describes the hardware characteristics of the end node, including its peripherals, sensors, and actuators. It provides context about the end node's capabilities and helps understand the services it can offer, guiding application development.
   - **Application TM:** This model describes *Interaction Affordances* offered by the application running on the end node. This model also ties in the previous two models, linking application logic to a specific *Embedded Core* firmware and a compatible hardware platform.

2. **Edge Node TMs**:

   Additionally, Edge Nodes are also characterized by the *Edge Connector TM* they implement, which reflects the capabilities of the *Edge Connector* service running on the edge node. These are also composable, allowing the following TMs to be defined:

- **Registration Listener TM:** Describes the API of the registration listener service running on the edge node. It defines the edge node's *Interaction Affordances* for registering new end nodes.
- **Node Controller TM:** Represents the capabilities of an end node controller service supported by the edge node. These models must be compatible with the *Embedded Core* firmware running on the end node it manages and with the registration listener service.
- **Edge Connector TM:** This model describes the overall capabilities of the edge connector service running on the edge node. It links to the registration listener and the node controller models, providing a complete overview of the edge connector instance. This model might also be extended with additional features related to the complementary features described in section 4.4.1, such as the *Web API* and *Proxy Service*.

Figure 4.12 illustrates the relationships between these models and how they are integrated into a cohesive framework. The nature of these relationships will be explored in the following sections, which provide a detailed overview of each model and its specific role within the *CityLink* architecture.

4.5.1 Application TM

As Figure 4.12 suggests, *Application TMs* are the central model for end nodes. Valid *Application TMs* consist of application-specific *Interaction Affordances* and link to both *Platform* and *Embedded Core* TMs. The goal for *Application TMs* is to provide *CityLink* with a clear definition of registered end nodes, by linking applications to compatible hardware and *Embedded Core* firmware and computational platforms.

This linking is essential because it not only bolsters linked data principles but also introduces a clear separation of concerns between the application logic and the underlying hardware and firmware, as well as promotes modularity and reusability.

This section is organized into three stages to facilitate understanding and introduce the structure of these models in a gradual and intuitive manner. First, a naive modeling attempt is presented to illustrate how an initial *Application TM* might be constructed, including its key components and the issues that arise when mixing abstraction levels. Next, a refined, protocol-agnostic version of the model is introduced. This version decouples the description of *Interaction Affordances* from concrete API endpoints and transport-layer details, providing a reusable base that focuses solely on the semantics of the application logic. Finally, a protocol-specific extension of the base model is developed, demonstrating how protocol bindings—such as MQTT topics—can be layered on top of the generic structure.

This modular, layered approach brings several advantages: it enables reusability of application definitions across different deployments, promotes portability by separating application logic from protocol bindings, and reduces duplication across models. Moreover, it avoids the pitfalls of

bloated TMs that attempt to support multiple protocol bindings simultaneously within the same document, making the overall modeling process more maintainable and scalable.

4.5.1.1 Naive Application TM

The following example describes a simple smart light application designed to illustrate the key concepts of an *Application TM*. The application allows users to check the current status of a light bulb, toggle it on or off remotely, and receive alerts if the light remains on for an extended period. While the scenario is minimal, it is a practical case study for introducing the structure and semantics of an *Application TM*.

The model will be assembled incrementally, proceeding from top to bottom, and each major component will be gradually examined. Starting with the root metadata and semantic context, we will define the main properties of the model. Then, we will explore how the model references the appropriate *Platform* and *Embedded Core TM*, establishing the hardware and firmware context in which the application runs. Finally, we will define the key *Interaction Affordances* that capture the behavior and interface of the application. This step-by-step approach aims to make the structure of the model and its rationale transparent while highlighting the purpose and usage of the most relevant vocabulary keys at each stage.

Context. Web of Things *Thing Models* and *Thing Descriptions*, as JSON-LD documents, start with the context definition.

```
"@context": [
  "https://www.w3.org/2022/wot/td/v1.1",
  "https://raw.githubusercontent.com/w3c/wot-binding-templates/refs
   ↪ /heads/main/bindings/protocols/mqtt/context.jsonld",
  "https://raw.githubusercontent.com/les2feup/CityLink/refs/heads/m
   ↪ ain/context.jsonld"
]
```

Listing 4.1: Application TM Context

Looking at Listing 4.1 line by line, the imported context definitions are:

1. The base Web of Things *Thing Description* context and vocabulary, as defined in [31].

2. The MQTT protocol bindings for the API endpoints that are defined later in the document. At the time of writing, these are not yet fully standardized and are only available on GitHub.

3. The *CityLink* context definition. Also available on GitHub [52], it defines the custom *CityLink* vocabulary terms that will be present throughout the document.

Type Annotations. JSON-LD documents may include type annotations, defining their particular nature. The WoT specification annotates TMs as `tm:ThingModel`. From the *CityLink* context extension, *Application TMs* must also be annotated with the `citylink:AppTM` type.

```
"@type": [
    "tm:ThingModel",
    "citylink:AppTM"
]
```

Listing 4.2: Application TM type annotations

Additional Top-Level Metadata. Beyond the mandatory fields, various other semantic annotations can be added at the top level of the document, either from the WoT vocabulary or external web ontologies. However, in the interest of simplicity and clarity, and given that no additional ontologies are imported in the context array defined above, only the *title*, *description*, and *version* keys are introduced in this step.

```
"title": "CityLink Light Control",
"description": "A simple application to control a light bulb.",
"version": "0.1.0"
```

Listing 4.3: Application TM title, description, and version keys

These keys provide human-readable metadata that helps developers and consumers understand the purpose and scope of the *Application TM*. Model titles should be globally unique to prevent ambiguity or collisions during discovery and reuse.

Furthermore, the *title* should remain unchanged across revisions, while the *version* key should be incremented to reflect updates, enabling version tracking and backward compatibility over time.

Links. The *Links* array is a mandatory field on *Application TMs* that must contain a link to the *Platform TM* describing the end node's hardware and the *Embedded Core TM* describing its firmware.

Examining Listing 4.4, the first entry pertains to the *Platform TM* and the second to the *Embedded Core TM*. In each *link* entry, four types of keys can be found:

- **rel** establishes the *link relation type*. As its name suggests, this entry reflects the relationship between the source resource (the *Application TM*) and the linked resource. In this case, the established relationship is `tm:submodel`, indicating that the linked resource is a TM to be included or merged into the current document.

```
"links": [
  {
    "rel": "tm:submodel",
    "href": "https://example.org/citylink/tm/platform/smart-light-b⌋
     ↪ ulb.tm.json",
    "type": "application/tm+json",
    "instanceName": "citylink:platform"
  },
  {
    "rel": "tm:submodel",
    "href": "https://example.org/citylink/tm/embedded-core/micropyt⌋
     ↪ hon-mqtt.tm.json",
    "type": "application/tm+json",
    "instanceName": "citylink:embeddedCore"
  }
]
```

Listing 4.4: Application TM links entry

- **href** represents the IRI of the linked resource. In the case of the *submodel* relationship, this IRI must be a resolvable URL so the resource can be fetched when needed.

- **type** establishes the content type of the linked resource. In this scenario, the `applic⌋ation/tm+json` content type serves as more of a formality, since the `tm:submodel` relationship established above already suggests that the linked resource should indeed be a Web of Things *Thing model*.

- Finally, **instanceName** serves two purposes: it acts as an identifier for the referenced *submodels* and defines the namespace prefix used when these models are merged. In this case, the values `citylink:platform` and `citylink:embeddedCore` are taken from the custom *CityLink* context.

  When the edge node merges the submodels into the final *Thing Description*, the *Interaction Affordances* imported from each will be prefixed with their respective *instanceName*. This approach enables efficient filtering and querying of *Interaction Affordances*, and it helps prevent naming collisions between similarly named properties, actions, or events originating from different models.

Citylink Manifest. Next, we define the *manifest* key. This field is a custom extension provided by the *CityLink* context, and is not part of the WoT vocabulary. It provides metadata required for deploying and instantiating the application on compatible end nodes.

This field contains two entries:

```
"manifest": {
    "placeholder": {
        "SOME_PLACEHOLDER": "some-value"
    },
    "source": [
        {
            "filepath": "main.py",
            "href": "https://example.org/citylink/application/cityl
             ↪  ink-light-control/source/main.py",
            "contentType": "text/x-python",
            "sha256": "{sha256-hash-of-the-file}"
        }
    ]
}
```

Listing 4.5: Application TM manifest entry

- **placeholder**: A map of custom placeholder values that can be substituted when generating a concrete *Thing Description* from this model. This allows developers to specify replacement values beyond the default placeholders recognized by the *CityLink* edge node.
- **source**: An array of objects, each describing a source file required by the application. Each object includes:
  - *filepath* – the destination path of the file within the end node's file system (if applicable);
  - *href* – a URL from which the edge node can download the file;
  - *sha256* – a SHA-256 hash of the file, used to verify its integrity after download;
  - *contentType* – the MIME type of the file, used to determine how the file should be handled or interpreted.

Alternatively, the entire **manifest** entry may be omitted from the model and provided via an external resource link in the *links* section using the relation type

```
"rel": "citylink:manifestLink"
```

This approach is advantageous when the manifest is large or varies across deployments, as it allows the same *Application TM* to be reused with different external manifests tailored to specific end nodes.

Interaction Affordances. The `"properties"`, `"actions"`, and `"events"` fields define the *Interaction Affordances* exposed by the application. These describe how external consumers, such as the edge node or user applications, can interact with the device at runtime.

Each *Interaction Affordance* includes a data schema that describes the structure and semantics of the exchanged data as well as a *forms* array with protocol-specific API endpoints. MQTT is used as the communication protocol in this example, and protocol-specific extensions from the WoT MQTT Binding specification [53] are applied.

These aspects will be made evident in the following examples.

Status Property. As stated at the beginning of this section, the application we are modeling allows users to check the current status of a light bulb. This is achieved in the *'Status' Property Affordance*. Under `"properties": {"status": { /*... */} }` we can find the contents of Listing 4.6.

```
"type": "boolean",
"description": "The current status of the light bulb.",
"readOnly": true,
"writeOnly": false,
"observable": true,
"forms": [
  {
    "href": "mqtt://broker.example.org:1883",
    "mqv:filter": "citylink/{{NodeID}}/light/status",
    "mqv:qos": 0,
    "mqv:retain": true,
    "contentType": "application/json",
    "op": [
      "readproperty",
      "observeproperty",
      "unobserveproperty"
    ]
  }
]
```

Listing 4.6: Application TM status property

This *Property Affordance* represents a boolean property indicating the light bulb's current operational state. The `"readOnly": true` field specifies that consumers cannot modify this property; it's a sensor-like value that reflects the actual hardware state of the light, updated internally by the system. Setting `"writeOnly": false` explicitly clarifies that the property is readable but not writable or write-only.

Additionally, the property is marked as `"observable": true`, allowing consumers to subscribe to updates and receive notifications whenever the light's state changes. This is especially useful for applications that require real-time synchronization, such as dashboards or mobile interfaces.

The `"forms"` field defines how the property is accessed over the MQTT protocol:

- **href** indicates the MQTT broker URL.
- The **mqv:filter** field sets the MQTT topic used to publish the property's value: `citylink/{{NodeID}}/light/status`.
- The `{{NodeID}}` placeholder is resolved by the edge node during Thing Description generation, ensuring topic uniqueness per device. This value will resolve to the end node's unique identifier generated during registration.
- The `"mqv:qos": 0` and `"mqv:retain": true` fields configure the delivery guarantees and message retention behavior at the broker level.
- **contentType** indicates the serialization format for the data received when reading this property.
- The **op** array lists the operations supported on this property: reading it once, subscribing to updates, and unsubscribing.

Together, these elements define a standard, read-only, observable interface that external consumers can use to monitor the state of the light in a reliable and protocol-aware manner.

Toggle Action. Under `"actions": {"toggle": { /*... */} }`, we define the *Action Affordance* responsible for changing the state of the light bulb. The full definition is shown in Listing 4.7.

This *Action Affordance* represents an executable command that allows consumers to turn the light on or off. The `"input"` field specifies the expected input schema: a boolean value where *true* corresponds to the "on" state and *false* to "off." The use of `"title"` and `"description"` inside the input schema ensures both human developers and consumers clearly understand the action's purpose.

The **forms** entry binds the action to a specific MQTT topic, `citylink/{{NodeID}}/light/toggle`, which the consumer must publish to to invoke the action. The `"op": "invokeaction"` field explicitly declares the operation type.

Unlike the status property, this action is not observable and does not support read or write operations. It is purely reactive and stateless: invoking it triggers a behavior on the device, without producing a direct response or altering the model. Any state change resulting from the action would be reflected in the *status* property, which can be observed separately.

```
"description": "Toggle the light bulb on or off.",
"input": {
  "type": "boolean",
  "title": "Toggle State",
  "description": "The desired state of the light bulb. 'true' for
   ↪  on, 'false' for off."
},
"forms": [
  {
    "href": "mqtt://broker.example.org:1883",
    "mqv:topic": "citylink/{{NodeID}}/light/toggle",
    "mqv:qos": 0,
    "mqv:retain": false,
    "contentType": "application/json",
    "op": "invokeaction"
  }
]
```

Listing 4.7: Application TM toggle action

Energy Alert Event. As the final piece of our example *Application TM*, we define an *Event Affordance* that captures excessive energy under `"events": {"energyAlert": {/**/}}`. The corresponding definition is shown in Listing 4.8.

This *Event Affordance* is designed to notify consumers when the light has remained on beyond a configured threshold, an energy-saving measure. The specifics of how the alert threshold is defined are irrelevant for this example. A simple approach, however, would be to include it as another property/action pair, as with the *status* property and the *toggle* action.

The `"data"` field defines the structure of the payload emitted with the event. It includes:

- **timestamp**: when the alert was triggered;
- **turnOnTime**: when the light was initially turned on;
- **duration**: how long the light has remained active, expressed in seconds.

The MQTT-specific binding in the `"forms"` array defines how clients can subscribe to or unsubscribe from this event. The use of `"mqv:filter"` ensures that messages are routed through a unique, per-device topic using the resolved `"{{NodeID}}"` placeholder.

The event is designed to be push-based, meaning notifications are sent only when triggered, without requiring polling from the client. This design suits resource-constrained networks well and enables responsive behavior in downstream applications.

```
"title": "Energy Consumption Alert",
"description": "An alert event that is triggered if the light is on
 ↪  for too long.",
"data": {
  "type": "object",
  "properties": {
    "timestamp": {
      "type": "string",
      "format": "date-time",
      "description": "The time when the alert was triggered."
    },
    "turnOnTime": {
      "type": "string",
      "format": "date-time",
      "description": "The time when the light was turned on."
    },
    "duration": {
      "type": "integer",
      "description": "The duration in seconds that the light has
       ↪  been on."
    }
  }
},
"forms": [
  {
    "href": "mqtt://broker.example.org:1883",
    "mqv:filter": "citylink/{{NodeID}}/light/alert",
    "mqv:qos": 0,
    "mqv:retain": false,
    "contentType": "application/json",
    "op": [
        "subscribeevent",
        "unbuscribeevent"
      ]
   }
]
```

Listing 4.8: Application TM energy alert event

Summary. This concludes the in-depth naive modeling exercise for an example *Application TM*, eligible for usage within the *CityLink* framework. The complete document, assembled by merging all snippets produced throughout this section, can be found under Appendix A, in Listing A.1.

As stated at the beginning of Section 4.5.1, this modeling approach—while easy to understand and sufficient for simple use cases—has notable limitations. Embedding application logic and protocol-specific bindings directly into a single document results in TMs that are harder to reuse, maintain, and adapt across different deployment scenarios. In the following sections, we address these issues by refactoring the TM into a more modular and protocol-agnostic form, promoting better separation of concerns and improved interoperability.

4.5.1.2 Protocol-Agnostic Application TM

To address the limitations discussed at the end of the previous section, we now refactor the light bulb model into a protocol-agnostic form. This modular version removes all protocol-specific bindings, separating the definition of *Interaction Affordances* from their concrete communication endpoints.

The transformation is straightforward: all **"forms"** entries are removed from the *properties*, *actions*, and *events* fields, as shown in Listing 4.9.

```
"properties": {
    "status": {
        "type": "boolean",
        "description": "The current status of the light bulb.",
        "readOnly": true,
        "writeOnly": false,
        "observable": true
        // -- Suppress the forms field --
    }
},
// Same for actions and events, suppressing the forms field
```

Listing 4.9: Protocol-agnostic property

Without these bindings, the model no longer depends on any particular transport protocol or serialization format. As a result, it also becomes unnecessary and inappropriate to include protocol-specific vocabularies (such as MQTT bindings) in the context array (Listing 4.10).

This version of the model also links to a more generic variant of the *Embedded Core TM*, one that similarly omits transport-specific details. This ensures semantic alignment and compatibility across submodels. While the *Platform TM* remains unchanged (since it does not include protocol bindings), the link to the *Embedded Core TM* is updated to reference a minimal, protocol-independent runtime specification (e.g., *micropython-base.tm.json*).

```
"@context": [
    "https://www.w3.org/2022/wot/td/v1.1",
    "https://raw.githubusercontent.com/les2feup/CityLink/refs/h
     ↪ eads/main/context.jsonld"
]
```

Listing 4.10: Protocol-agnostic Context

Lastly, the **manifest** field is replaced by a `citylink:manifestLink`, referencing an external manifest file that holds the same metadata and source information. This keeps the base model lightweight and more adaptable to different deployment scenarios.

```
"links": [
    {/*Same platform model as before*/},
    {
        "rel": "tm:submodel",
        "href": "https://example.org/citylink/tm/embedded-core/
         ↪ micropython-base.tm.json",
        "type": "application/tm+json",
        "instanceName": "citylink:embeddedCore"
    },
    {
        "rel": "citylink:manifestLink",
        "href": "https://example.org/citylink/application/light
         ↪ -control/manifest.json",
        "type": "application/json"
    }
]
```

Listing 4.11: Protocol-agnostic link entries

This refactoring improves reusability, facilitates cross-protocol interoperability, and encourages better separation of concerns. Concrete protocol bindings can later be layered on top of this base model without modifying its core semantics, as explored in the next section.

As with the previous example, the complete *Application TM* resulting from this transformation can be found in Appendix A, under Listing A.2.

4.5.1.3 Protocol-Specific Application TM

Extending the application logic with concrete protocol bindings for a specific deployment becomes possible once the application logic is captured in a clean, protocol-agnostic model, extending it with concrete protocol bindings for a particular deployment becomes possible. This is done by layering protocol-specific definitions on top of the base model, resulting in a complete *Application TM* that is aligned with a specific *Embedded Core* implementation.

First, we re-import the protocol bindings that were removed from the previous model into the context, resulting in the same `@context` definition previously presented in Listing 4.1.

Next, we update the **links** section, as shown in Listing 4.12. First, we introduce a *tm:extends* relation to the protocol-agnostic base model defined in the previous section. This signals to the edge node that all definitions and metadata from the referenced model should be inherited into the current one.

In addition, we override the `citylink:embeddedCore` link to point to the MQTT-enabled *Embedded Core TM*, matching the runtime used in the original, naive *Application TM*. This ensures compatibility between the extended application logic and the communication bindings implemented by the end node's firmware.

```
"links": [
    {
      "rel": "tm:submodel",
      "href": "https://example.org/citylink/tm/embedded-core/micr⌋
       ↪  opython-mqtt.tm.json",
      "type": "application/tm+json",
      "instanceName": "citylink:embeddedCore"
    },
    {
      "rel": "tm:extends",
      "href": "https://example.org/citylink/tm/application/cityli⌋
       ↪  nk-light-control-base.tm.json",
      "type": "application/tm+json",
    }
]
```

Listing 4.12: Protocol-specific link entries

Finally, we can return the **forms** entries to our *Interaction Affordances* (Listing 4.13, so that after merging both models, the resulting document is equivalent to our first naive modeling attempt.

This compositional structure greatly improves modularity and reuse. Instead of rewriting or duplicating complete TMs for each target configuration, developers can maintain slim, focused

```
"properties": {
  "status": {
    "forms": [
      {
        "href": "mqtt://broker.example.org:1883",
        "mqv:filter": "citylink/{{NodeID}}/light/status",
        "mqv:qos": 0,
        "mqv:retain": true,
        "contentType": "application/json",
        "op": [
          "readproperty", "observeproperty", "unobserveproperty"
        ]
      }
    ]
  }
}
// Same for actions and events...
```

Listing 4.13: Protocol-specific Interaction Affordances.

base models and layer bindings as needed. It also supports better tooling: models can be programmatically extended based on deployment targets, enabling automatic code generation or drag-and-drop configuration workflows in graphical editors.

Compared to the naive/monolithic model (Listing A.1), this approach cleanly separates concerns. Application logic is authored once, while protocol bindings are swapped depending on the chosen communication stack. This separation is a key design principle of the *CityLink Relational TM Schema*, promoting adaptability, interoperability, and long-term maintainability across heterogeneous IoT deployments.

Finally, to consult the complete TM resulting from this approach, refer to Appendix A, Listing A.3. The following sections will cover the remaining TM types defined in the *CityLink* framework more concisely, as the core modeling principles and patterns introduced here—particularly the layered, modular approach—apply similarly across all model types.

4.5.2 Embedded Core TM

Embedded Core TMs describe the firmware-level functionality of *CityLink* end nodes. Like *Application TMs*, they are composable: an implementation family can be defined as a group of protocol-specific variants extending a common, protocol-agnostic base model. This modularity allows shared behavior and core functionality to be defined once and reused across multiple bindings and firmware targets.

An *EmbCTM* typically defines the API exposed by the *Embedded Core* firmware, namely, properties, actions, and events used for registration, device control, and runtime adaptation. These

Interaction Affordances reflect the internal capabilities of the firmware and should cover the functions required by the edge node to manage the device lifecycle.

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        "https://example.org/CityLink/context.jsonld"
    ],
    "@type": [ "tm:ThingModel", "citylink:EmbCTM" ],
    "name": "MicroPython MQTT Embedded Core",
    "description": "A MicroPython-based embedded core for CityLink
     ↪  end nodes with MQTT support.",
    "links": [
        {
            "rel": "tm:extends",
            "href": "https://example.org/citylink/tm/platform/micro
             ↪  python-base.tm.json",
            "type": "application/tm+json",
        },
        {
            "rel": "citylink:controlledBy",
            "href": "https://example.org/citylink/tm/edge-connector
             ↪  /micropython-mqtt-node-controller.tm.json",
            "type": "application/tm+json",
        },
        {
            "rel": "citylink:manifestLink",
            "href": "https://example.org/citylink/embedded-core/mic
             ↪  ropython-mqtt/manifest.json",
            "type": "application/json"
        }
    ],
    // interaction affordances
}
```

Listing 4.14: Example of a partial Embedded Core TM

The key components of this model are highlighted in Listing 4.14, which shows a valid *Embedded Core TM* for a MicroPython-based firmware with MQTT support.

As with other *CityLink TMs*, *Embedded Core TMs* declare their type explicitly using `citylink:EmbCTM` and may extend a base model (e.g., *micropython-base.tm.json*) using the *tm:extends* relation.

Crucially, these models also include a `citylink:controlledBy` link to a compatible *Node Controller TM*, indicating which controller service should be launched on the edge node after device registration. This connection, illustrated in Figure 4.12, is essential for linking embedded

behavior with external orchestration.

As with application models, an *Embedded Core TM* must also define either an inline `city_link:manifest` field or a `citylink:manifestLink` pointing to a manifest file describing the firmware source or binary. However, unlike application code—which may be updated more frequently—*Embedded Core* firmware is only updated when explicitly required by an adaptation request.

This separation is deliberate. It allows lightweight application-level updates without redeploying the entire firmware image. In interpreted environments like MicroPython, where the *Embedded Core* and application logic reside in the file system, granular updates are easily achieved by overwriting only the affected scripts.

In contrast, compiled firmware (e.g., written in C or Rust) typically requires full binary image deployment. In these cases, the manifest defined in the *Embedded Core TM* may be omitted or its role delegated to the application manifest if it contains the complete build output.

Ultimately, the update and deployment strategy, along with the content and usage of the *Embedded Core TM* manifest, will vary depending on the characteristics of the target runtime and the constraints of the deployment environment.

4.5.3 Platform TM

The *Platform TM* is the third and final type of TM defined specifically for end nodes within the *CityLink* framework.

The purpose of the *Platform TM* is to describe the hardware characteristics of the end node, including available peripherals, sensors, actuators, and other physical interfaces. Its primary role is to enrich the end node's metadata with information about its concrete capabilities. This enables consumers (human developers or automated tools) to reason about the device's functional potential and guide application development or deployment.

If described in sufficient detail, a *Platform TM* can serve as input to code generation tools that automatically generate application scaffolding based on the defined hardware and the corresponding logic expressed in the *Application TM*. Furthermore, these models enable compatibility checks to be performed at runtime or deployment time, allowing the system to verify whether a given application can be executed on a specific device based on its hardware profile.

However, at the time of writing, there is no standardized vocabulary or schema within the W3C WoT ecosystem suitable for describing hardware platforms at this level of detail. As a result, the practical use of *Platform TMs* in this dissertation is limited to providing contextual metadata about the end node's physical capabilities. In the `CityLink` framework, such models are identified by including the type annotation `citylink:PlatTM`.

Developing a robust and widely accepted ontology for hardware representation is a substantial standardization effort in its own right. While it would significantly enhance the utility of *Platform TMs*, addressing this challenge falls outside the scope of the present dissertation work.

4.5.4 Edge Node TMs

On the edge node side, the three model types presented in Figure 4.12 describe the capabilities of the *Edge Connector* service. A single edge node may host multiple *Edge Connector* instances, each with its own *Registration Listener* and a set of *Node Controller* implementations, enabling it to manage a heterogeneous set of end nodes using different communication protocols.

As with the other TMs, each *Edge Connector TM* must declare the type `citylink:ECTM` and can be composed by linking to modular subcomponents using the appropriate relations.

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        "https://example.org/CityLink/context.jsonld"
    ],
    "@type": [ "tm:ThingModel", "citylink:ECTM" ],
    "name": "MicroPython MQTT Edge Connector",
    "description": "An edge connector for CityLink end nodes with
     ↪ MicroPython and MQTT support.",
    "links": [
        {
            "rel": "tm:submodel",
            "href": "https://example.org/citylink/tm/edge-connector
             ↪ /mqtt-registration-listener.tm.json",
            "type": "application/tm+json",
            "instanceName": "citylink:regListener"
        },
        {
            "rel": "citylink:hasController",
            "href": "https://example.org/citylink/tm/edge-connector
             ↪ /micropython-mqtt-node-controller.tm.json",
            "type": "application/tm+json",
        }
        // other links to node controllers
    ]
}
```

Listing 4.15: Example of a partial Edge Connector TM

Listing 4.15 shows a valid *Edge Connector TM* for a MicroPython-based edge node with MQTT support. The structure uses well-defined link relations to establish the model composition as in previous examples.

Each *Edge Connector TM* must include:

- A link to its associated *Registration Listener TM* using the `tm:submodel` relation. This link must also declare the `instanceName` as `citylink:regListener`, ensuring consistent namespace expansion and behavior during model merging.

- One or more links to supported *Node Controller TMs* using the custom relation `citylink:hasController`. These indicate which runtime controller modules the Edge Connector can instantiate to manage different types of end nodes.

At least one *Node Controller* linked in the model must be compatible with the associated *Registration Listener* since the registration process relies on their interaction. If no compatible Node Controller is available, the edge node cannot manage any end nodes, rendering the Edge Connector non-functional for its intended purpose.

To ensure correct classification and validation:

- *Registration Listener TMs* must declare the type `citylink:RegLTM` and define a *'registration' Action Affordance* representing the expected registration input, as discussed in Section 4.2.2.
- *Node Controller TMs* must carry the type `citylink:NCTM` and expose the runtime capabilities required to configure and manage specific types of end nodes once they are registered.

4.6 Conclusion

This chapter has presented the *CityLink* framework, a novel approach to managing end nodes in urban IoT deployments by leveraging and extending the W3C WoT Thing Description standard. In particular, the *Relational TM Schema* has been introduced, which provides a standard data model and API contract for end nodes, edge nodes, and consumers. The framework is built around the concept of *Thing Models*, which are composable and allow physical devices to be described modularly, promoting reusability and interoperability. *CityLink* promotes the semantic interoperability of data via the linked-data paradigm, which also opens the way for powerful search and discovery capabilities, allowing consumers to find and interact with end nodes based on their capabilities and application logic using the W3C WoT Discovery specification. The framework also integrates modern features such as automated registration of new nodes in the IoT system, over-the-air updates, and remote adaptation of end nodes, allowing for dynamic changes to the application logic and capabilities of the end nodes without requiring physical access to the devices. The framework supports a wide variety of communication technologies, allowing for easy extension of the proposed data models and the addition of new protocols and technologies as they become available. This flexibility provides a solid foundation for building scalable and adaptable urban IoT systems that can evolve to meet the changing needs of cities and their inhabitants. Furthermore, it allows for possible deployment scenarios other than the envisioned Smart City use case, such as Smart Buildings, Smart Homes, and other IoT applications that require a flexible and adaptable approach to managing end nodes in a heterogeneous environment.

Chapter 5

CityLink Implementation Walkthrough

Following the conceptual and architectural exposition presented in Chapter 4, this chapter outlines a structured walkthrough of the implementation process for the *CityLink* framework. Its purpose is twofold: first, to provide a high-level procedural guide for implementing *CityLink*-based systems, and second, to contextualize the specific proof-of-concept deployment described in the following chapter.

Two illustrative implementation scenarios are presented. The first outlines a generalized, step-by-step approach for building a *CityLink* deployment from the ground up, assuming no pre-existing components are available or applicable. This approach is particularly relevant for customized deployments or use cases that fall outside the scope of any pre-existing implementations.

The second scenario applies the same logic while leveraging the components developed for the proof-of-concept deployment carried out as part of this dissertation. This example demonstrates how the framework can be adapted and instantiated using existing software modules and system configurations.

Together, these examples provide both a practical implementation guide and a conceptual bridge to Chapter 4, which presents the technical design, operational behavior, and performance evaluation of the implemented components. As such, this chapter also serves as a roadmap for developers, summarizing the key steps involved in adopting or extending the CityLink framework in real-world applications.

5.1 Overview

The flowchart depicted in Figure 5.1 provides a high-level overview of the decision-making process and the primary implementation steps involved in adopting the *CityLink* framework. Each node in the diagram has been color-coded to convey its functional role: red nodes indicate the start and end points; blue nodes represent inputs or system requirements that drive the process; green diamonds correspond to decision points; and yellow rectangles denote specific implementation tasks.

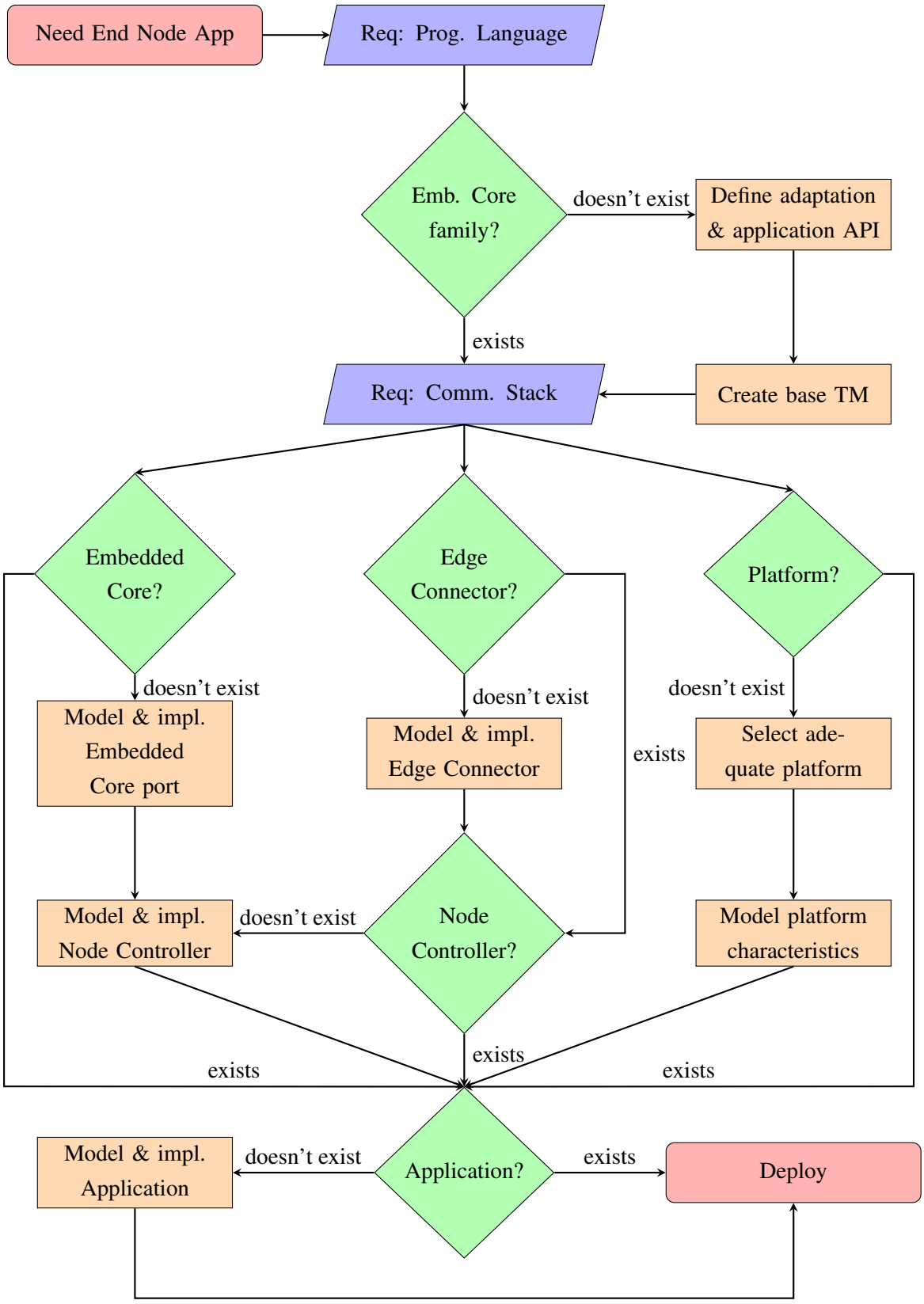


Figure 5.1: CityLink implementation process flowchart diagram

While the diagram concisely represents the framework's implementation logic, specific details have been intentionally omitted to maintain visual clarity and accessibility. These omitted elements are elaborated upon in the subsequent sections.

The process begins at the node labeled *Need End Node App*, which assumes an application or functionality must be executed on one or more end nodes. From this point, the flowchart guides the implementer through a sequence of tasks and decisions culminating in a fully deployed CityLink-based system, with the target application (alongside any supporting components) made operational and accessible to consumers.

5.2 Ground-Up Implementation

This section considers a clean-slate scenario in which the *CityLink* framework is to be implemented entirely from scratch. We assume there are no pre-existing deployments of end nodes or edge nodes, and no reusable software components are available at any architecture layer.

This exercise outlines a complete instantiation of the *CityLink* software stack, progressively building up the necessary elements, such as TMs, communication components, and deployment configurations, from the ground up. At each decision point within the implementation flowchart, we follow the path that assumes the relevant component does not yet exist, thereby constructing a complete and self-contained deployment pathway.

5.2.1 Stage 1: Define End Node Requirements

The implementation process begins with a functional requirement: the need for one or more end nodes to execute a specific application. This requirement is the entry point to the *CityLink* framework's deployment process.

The first technical input involves selecting the end node's target programming language and runtime environment. This choice determines whether an existing *Embedded Core* implementation can be reused. In this ground-up scenario, we assume that no compatible *Embedded Core family* exists for the chosen language or runtime. As a result, a new **API specification** must be created to define how applications interact with the runtime. This involves defining two key interfaces:

- **Internal API (Application Interface):** This interface specifies how end-user applications interact with the underlying runtime. It includes:
  - Mechanisms for declaring and handling WoT *Interaction Affordances* (properties, actions, events),
  - Task scheduling primitives (e.g., timers, message callbacks)
  - Utility functions required by applications (e.g., logging, state management, data marshaling).
- **External API (Adaptation Interface):** This interface defines how the runtime presents its *Interaction Affordances* to the rest of the system for adaptation and configuration purposes.

These affordances are formally described in a base TM, a shared contract that all implementations of this *Embedded Core* family must conform to, regardless of the underlying communication stack or language-specific details.

By separating internal and external concerns, the framework promotes modularity and reusability. The internal API ensures that application logic can be developed independently of communication details, while the external API, modeled semantically through the TM, enables consistent integration with the edge-layer services responsible for registration, configuration, and adaptation.

This stage establishes the foundational interface definitions necessary to implement a new Embedded Core family, enabling subsequent development of the runtime, its semantic model, and the applications that will run on top of it.

5.2.2 Stage 2: Establish Communication Requirements

Once the application and runtime interface have been defined, the next step is determining the system's communication requirements. The second input node captures this, *Req: Comm. Stack*, which specifies the desired network and messaging protocols for data exchange between end nodes and the edge infrastructure.

The communication stack typically builds upon IP-based technologies. Web-based protocols such as HTTP, CoAP, and MQTT are particularly well-suited for integration with the *Web of Things* paradigm, as their structures map naturally to WoT interaction patterns (properties, actions, and events). These protocols are natively supported by most web-based and embedded platforms, being popular choices for IoT applications.

Nevertheless, *CityLink* is designed to be extensible. Using semantic annotations, protocol bindings, or the introduction of intermediary gateway devices, alternative communication stacks, including those used in constrained or industrial environments, can also be integrated into the framework.

With the communication stack defined, three critical components must now be addressed:

- The *Edge Connector* implementation, including a *Registration Listener* and *End Node Controller* compatible with the *Edge Connector*.
- Selecting an adequate computational platform for the end node and creating a suitable *Platform TM* describing its characteristics.

In this ground-up scenario, we assume that none of these components currently exist. Consequently, each must be modeled and implemented from scratch, as described in the next stage.

5.2.3 Stage 3: Develop Core Components

With the communication stack defined, three critical components must now be developed to enable interaction between end nodes and the edge node infrastructure:

1. **Model and Implement the Embedded Core Port**

   A new port of the *Embedded Core* must be designed and implemented for the selected programming language and runtime environment. This includes implementing the application and adaptation APIs defined earlier and supporting the chosen communication protocols.

   Furthermore, the *Embedded Core* must act as a client of the registration API provided by the corresponding *Edge Connector* service. This allows each end node to announce its presence and expose its affordances to the edge infrastructure.

   A protocol-specific extension of the previously defined base *Embedded Core TM* must also be created. This extended model includes protocol bindings, security schemes, and network endpoints specific to the new *Embedded Core* port.

2. **Model and Implement the Edge Connector**

   A compatible *Edge Connector* must be developed on the edge node side to handle registration requests over the selected communication protocol. As part of this step, a registration API must be defined and documented for use by all end nodes based on the chosen protocol stack.

   Optional but recommended components such as the *CityLink Web API* and the *Proxy Service* may also be implemented at this stage. These provide interfaces for monitoring, debugging, and integrating the edge node with higher-level services or user interfaces.

3. **Model and Implement the End Node Controller**

   Finally, a *Node Controller* compatible with the developed *Embedded Core* and its adaptation API must be implemented. The controller is deployed alongside the *Edge Connector* and handles adaptation requests, mediating interactions, and potentially proxying communication to and from the registered end nodes.

   The *Node Controller* must be designed in close coordination with its corresponding *Embedded Core*, as they are closely coupled elements of the systems. Additionally, the *Embedded Core TM* for the new *Embedded Core* port must link to the *Node Controller TM* of its matching controller.

5.2.4 Stage 4: Define Platform Characteristics

Platform-specific considerations must also be addressed in parallel with the development of core runtime components. Since no suitable *Platform TM* exists for the target end node hardware in this ground-up scenario, the following steps must be undertaken:

- **Select an Adequate Platform**

  The developer must select a suitable computational platform compatible with the developed *Embedded Core* firmware and appropriate for the intended application scenario. This typically involves choosing a microcontroller, provisioning network connectivity (e.g., Wi-Fi, cellular, LoRa, etc), and integrating additional components such as sensors and actuators as the use case requires.

- **Create a *Platform TM***

  Once the hardware platform is selected, a corresponding *Platform TM* must be created. This model semantically describes the capabilities and constraints of the platform, including available connectivity options, computational resources, sensing and actuating capabilities, power limitations, and other relevant features. This model can then be used to support downstream tasks such as deployment automation, compatibility checking, and adaptation logic at the edge layer.

5.2.5 Stage 5: Finalize Application and Deployment

All required runtime components and corresponding **TMs** have been developed at this stage. The final task is to implement the application logic that fulfills the functional requirements identified at the beginning of the process.

Since we assume no suitable application is available, an *Application TM* must be created. This model describes the desired *Interaction Affordances*, such as properties, actions, and events, that the end node is expected to expose to consumers.

Once defined, the *Application TM* serves as a blueprint for implementing the application logic using the application API provided by the *Embedded Core* runtime. Because the internal API was standardized earlier, this translation from model to code can be performed consistently and systematically.

With the application implemented, the system is ready for deployment. In the context of the *CityLink* framework, deployment encompasses several coordinated steps:

- **Publishing Thing Models:** All TMs created during the development process, including *Platform*, *Embedded Core*, *Node Controller*, and *Application TMs*, must be made accessible to the edge node. This can be achieved through various mechanisms, such as a dedicated web server, an RDF store or semantic database [24], [25], a version-controlled repository (e.g., Git), or by hosting them locally on the edge node.

- **Configuring and Deploying the End Node:** The end node must be programmed with the appropriate firmware, including the *Embedded Core*, the application code (optionally), and any hardware-specific drivers or configurations. It must also be provisioned with network credentials and any necessary runtime parameters.

- **Configuring and Deploying the Edge Node Stack:** The edge node must be set up with the *Edge Connector* service and the corresponding *Node Controller*. Any optional services, such as the *CityLink Web API* or data processing pipelines, can also be configured at this stage.

This completes the ground-up implementation process for the *CityLink* framework. The approach ensures that each layer—from hardware to semantics—is developed in a modular, reusable, and standards-aligned manner, supporting a broad range of IoT application scenarios.

5.3 Implementation Using the Proof-of-Concept Components

The proof-of-concept deployment developed for this dissertation provides a functional set of *CityLink* components that can be reused and extended for rapid prototyping and application development. The following core components have been implemented:

- **Embedded Core:** MicroPython-based runtime with MQTT communications over Wi-Fi.
- **Edge Connector:** MQTT-compatible registration handler.
- **Node Controller:** MQTT & MicroPython-compatible controller for managing end node adaptation and proxying.
- **Platform TM:** Base TM for the Raspberry Pi Pico W.

Additionally, the edge node stack includes optional but useful extensions:

- **Web API:** Partial WoT *Thing Description Directory* extended with CityLink-specific endpoints.
- **Web Interface:** Basic user interface for browsing registered nodes and *Thing Descriptions*.

5.3.1 Deployment Workflow

A developer wishing to deploy a new MicroPython + MQTT application using these components can follow a streamlined process:

1. **Platform Modeling:** Either reuse the existing *Platform TM* for the Pico W, extend it to include additional peripherals or characteristics, or model a new platform, provided it is MicroPython-compatible.
2. **Application Modeling:** Define a new *Application TM*, or extend an existing one, to describe the desired interaction affordances.
3. **Application Development:** Implement the application logic using the API exposed by the MicroPython *Embedded Core* runtime.

4. **Thing Model Publication:** Publish the new *Platform TM* and *Application TM* to the official *CityLink* repository on GitHub [52], or to a custom *Thing Model* server.

5. **End Node Deployment:** Flash the end node with the Embedded Core firmware and application code. Configure the device with Wi-Fi credentials, MQTT broker settings, and a reference to the *Application TM* for self-identification and adaptation.

6. **(Optional) Pre-Adaption:** The application code may be pre-programmed with the desired affordances, skipping the initial adaptation step after registration.

This process enables rapid deployment of new applications by reusing and extending the proof-of-concept components, significantly reducing the effort required for initial integration.

5.4 Conclusion

This chapter has provided a practical walkthrough of the implementation process for systems based on the *CityLink* framework. Two distinct paths were explored: a comprehensive, ground-up development scenario and a simplified deployment example leveraging the proof-of-concept components developed for this dissertation.

The ground-up example serves as a complete blueprint for developers starting from scratch. It outlines each required step, from API specifications and TM design to software development and deployment, demonstrating the modular and extensible nature of the framework.

To complement this, the second example showcased a shortened development cycle process using the ready-made components developed as part of this work's proof-of-concept. This approach significantly reduces development effort and illustrates the practical value of reusable software modules, TMs, and optional services such as the *CityLink Web API* and user interface.

Table 5.1 summarizes the key artifacts involved in the proof-of-concept deployment, categorizing each element by its role, function, and level of reuse. This overview reinforces the modular architecture of the *CityLink* framework and highlights the benefits of its model-driven design approach.

As outlined in the introduction, the purpose of this chapter has been to bridge the gap between the conceptual and architectural principles introduced in Chapter 4 and the detailed, implementation-oriented discussion presented in the next chapter.

Chapter 6 will expand on the brief overview provided in this chapter, presenting an in-depth look at the proof-of-concept implementation. It will delve into each component's technical design and development, offering a comprehensive evaluation of the system architecture, tooling choices, and the rationale behind key engineering decisions.

By introducing both the methodology and the supporting tools required to deploy *CityLink*-based systems, this chapter lays the groundwork for a deeper understanding of the implementation process. It contextualizes the following discussion and equips the reader to critically assess the framework's practicality, flexibility, and extensibility in real-world scenarios.

Table 5.1: Artifacts used in the Proof-of-Concept Deployment

Artifact	Type	Purpose	Usage
Core Runtime and Edge Services			
MicroPython & MQTT Embedded Core	Runtime Component	Provides application and adaptation APIs, handles MQTT communication	Reused
MQTT Edge Connector	Edge Service	Handles end node registration and TM resolution	Reused
MicroPython & MQTT Embedded Core	Edge Component	Manages adaptation and proxies affordances to the edge node	Reused
Thing Models			
Embedded Core TM	Thing Model	Describes the external API and adaptation interface exposed by the Embedded Core	Reused
Edge Connector TM	Thing Model	Describes the registration API and coordination capabilities of the Edge Connector	Reused
Node Controller TM	Thing Model	Describes the adaptation controller's affordances and behavior	Reused
Pico W Platform TM	Thing Model	Describes the hardware characteristics of the Raspberry Pi Pico W	Reused or Extended
Application TM	Thing Model	Defines the application's semantic interface (affordances)	New
Application and Supporting Infrastructure			
Application Code	Firmware Module	Implements logic corresponding to the Application TM using Embedded Core API	New
CityLink Web API	Web Interface	Serves *TMs and Descriptions*, registration info, and adaptation status through a RESTful interface	Optional (Reused)
CityLink Repository	Information Storage / Database	Central location for hosting TMs (e.g., GitHub or dedicated service)	Reused / Updated

Chapter 6

Results

A proof-of-concept implementation of the proposed framework was developed using a single edge node managing a set of real and simulated end devices running prototype versions of the software and data model components as described in chapter 4. Figure 6.1 presents a deployment diagram of the proof-of-concept scenario. The source code and reproduction instructions are available in a GitHub repository [52].

Using MicroPython to enable a faster development cycle, a protocol-agnostic *Embedded Core TM* was created, defining the general interface for a MicroPython *Embedded Core* implementation family. The base model was then extended to create a specific implementation for MicroPython end nodes communicating via MQTT over a Wi-Fi network. Applications developed for the prototype *Embedded Core* runtime are developed against the general API defined by the base model, enabling easy migration to other protocol implementations in the future, such as CoAP or HTTP. To test the prototype implementation, the code was deployed alongside the MicroPython firmware on a Raspberry Pi Pico W, which is a low-cost microcontroller board with built-in Wi-Fi capabilities and a dual-core RP2040 System on Chip (SoC) [54].

On the edge node, an Eclipse Mosquitto [55] instance was deployed alongside an *Edge Connector* service implemented using the TypeScript language. Matching the prototype end node, the developed *Edge Connector* service implements an MQTT client for listening to incoming registration requests (Figure 6.1, *Registration Listener* module). A *Node Controller* module compatible with the *Embedded Core* API was also implemented, enabling the edge node to manage adaptation requests targeting the prototype end node.

Extending the *Edge Connector* service, a *Thing Description Directory* (TDD) component was implemented to assume the role of the *Web API & Proxy Service* described in chapter 4. This component implements the *Things, Search and Events APIs* as described in the *Thing Discovery* specification [32], with the limitations to the CRUDL operations described in section 4.4.1. Alongside the discovery APIs, the *Thing Description Directory* also implements a set of APIs supporting the additional features provided by the *CityLink* framework, such as end node adaptation and thing model management. A simple web interface was also developed to allow consumers to easily interact with the *Edge Connector* service and the TDD.

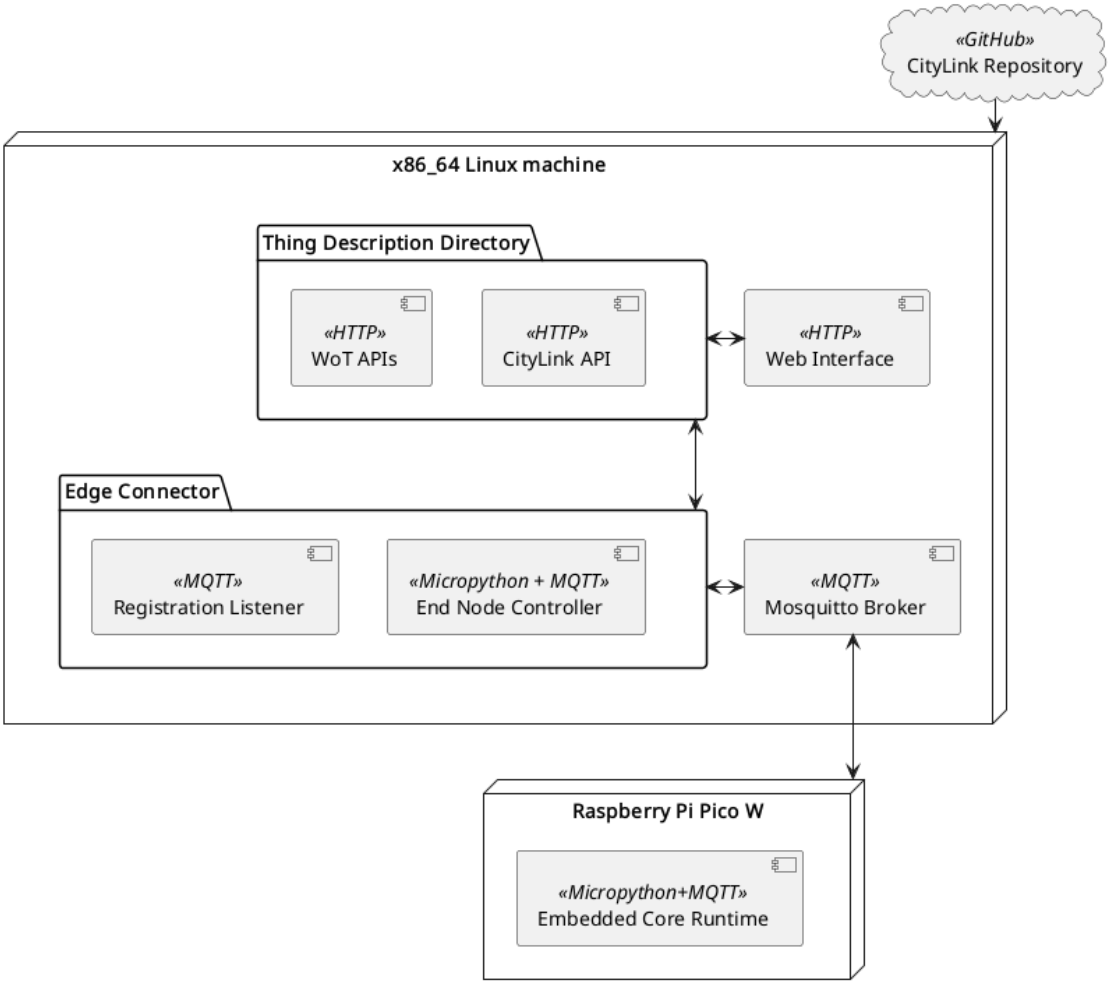


Figure 6.1: Proof-of-concept deployment over a WLAN with MQTT.

Finally, the *CityLink Repository* on GitHub [52] was used to store TMs and application code, allowing for easy access and versioning. Libraries from the Eclipse ThingWeb project [49] were utilized to facilitate the manipulation of TMs and ensure the correct instantiation of *Thing Descriptions*.

This software stack was deployed on a standard x86_64 machine running the SMP PREEMPT_DYNAMIC 6.14 Linux kernel, which served as the edge node, hosting the MQTT broker and the *Edge Connector* service. With containerization, simpler ARM machines, such as a Raspberry Pi, can be the *Edge Node*, allowing for a more cost-effective deployment solution.

MQTT was chosen as the communication protocol due to its widespread usage in the IoT and to demonstrate its useful features for systems and urban IoT deployments in general. As mentioned previously, edge nodes may proxy the *Interaction Affordances* of end nodes, reducing the number of requests directly handled by constrained devices and possibly enhancing the overall system's responsiveness. While this would typically be directly integrated into the *Edge Connector* functionality or offloaded to a dedicated custom-made component hosted by the edge node, the

broker can manage this directly with MQTT.

Using the MQTT *retain* flag, the broker can cache the most recent values of an end node's properties for on-demand consumer access. Furthermore, events are automatically forwarded to all subscribers due to the nature of the protocol and can also be optionally cached. The broker manages any parallelization of outgoing requests, not the publishing device. Finally, action invocations are forwarded from consumers to the respective end node in a FIFO manner. The constrained device only needs to communicate with the broker and does not need to respond to any external connections.

6.1 Thing Models

As mentioned, the *CityLink* framework relies on TMs to define the data model and API of the end nodes. For the proof-of-concept, various TMs were created to represent the different components of the deployed IoT system, including test end nodes, the edge node, the *Edge Connector* service, and the *Embedded Core* runtime. In defining these models, the *import* and *extension* features were extensively used to create a hierarchy of models, allowing for the reuse of standard definitions and the extension of existing models to make more specific ones. All models are available in the *CityLink Repository* on GitHub [52].

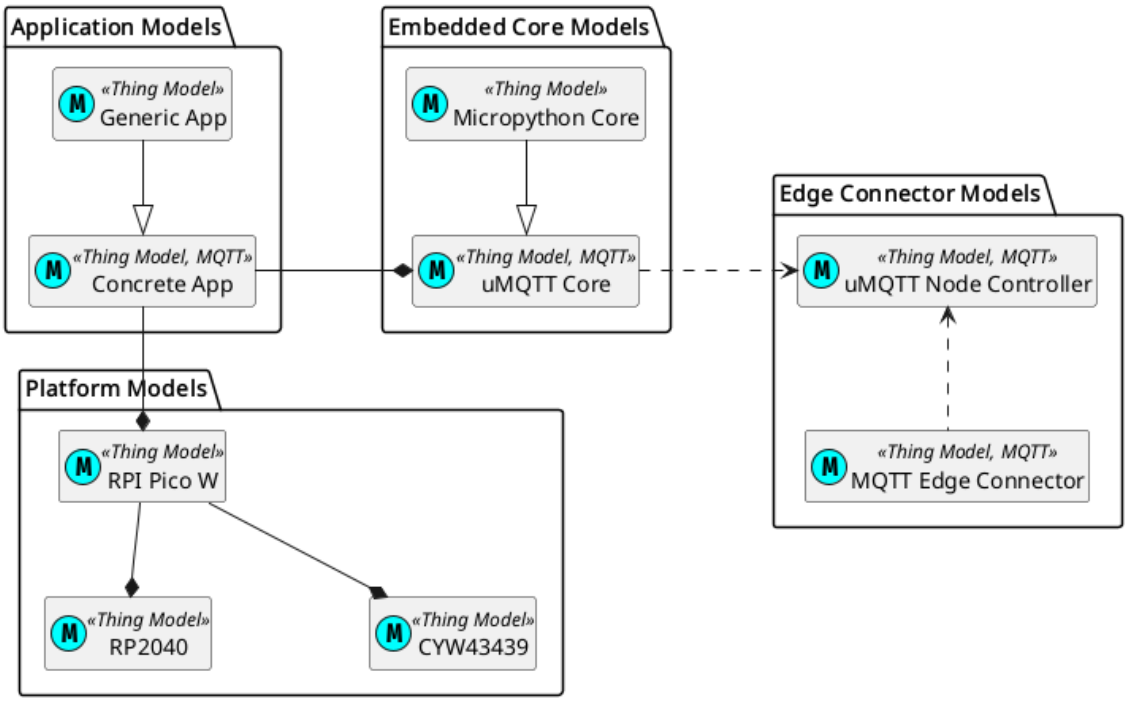


Figure 6.2: Relationships between various *Thing Models* used in the proof-of-concept.

Figure 6.2 presents a diagram of the relationships between the various TMs used in the proof-of-concept. For convenience, *Application TMs* are not directly included in the diagram, with

their relationships to the remaining models being represented by the catch-all *Generic APP* and *Concrete APP* components.

Looking at the diagram, *tm:submodel* relations are represented by filled lines terminating in a lozenge shape (⧫). These indicate that the given model is a collection of submodels. This was used in the *RPI Pico W* [56] model, which aggregates the *RP2040* SoC and the *CYW43439* network controller models. Concrete *Application TMs* also reference an *Embedded Core TM* as a submodel, as established in chapter 4, creating a logical connection between the application and the runtime it is designed to run on.

Extension relationships are represented by filled lines terminating in a triangle shape (△). These indicate that a given TM directly builds on the definitions of another, providing additional definitions or overriding existing ones. This is used by concrete *Application TMs* to enrich their generic counterparts with the protocol-specific definitions required for consumers to interact with the application. Similarly, the *uMQTT Core TM* (defined for the MicroPython & MQTT *Embedded Core* implementation) extends the *MicroPython Core TM* created as a basis for the *Embedded Core* family.

Finally, indicated by the dashed lines, other link relationships were used to indicate relationships between TMs models in the *CityLink* framework that fall outside the WoT *Thing Description* [31] specification. As per Chapter 4, these relationships include a link between *Edge Connector TMs* and *Node Controller TMs* using the *citylink:hasController* relation (see Figure 6.2, connection between *uMQTT Node Controller* and *MQTT Edge Connector*). The other link relationship is used to connect the *uMQTT Core TM* to its matching controller, the *uMQTT Node Controller TM*, using the *citylink:controlledBy* relation.

6.2 Embedded Core Implementation

Refered to as *uMQTT Core* in short, the *Embedded Core* implementation for the proof-of-concept is based on the MicroPython programming language and the MQTT protocol. MicroPython is a lean Python 3 implementation designed to run on microcontrollers and in constrained environments [50].

The structure of the *uMQTT Core* implementation is roughly similar to the one described in Figure 4.7 of chapter 4, repeated here for convenience.

The *uMQTT Core* implementation is divided into four main components:

- **Boot File**: The entry point for the MicroPython interpreter, responsible for initializing the *Embedded Core* runtime.
- **Core Module**: The main module implementing all the essential affordances as defined in figure 6.3.
- **Extension Modules**: Additional modules loaded at runtime to extend the runtime with the necessary API for applications to interact with the end node. Included here are the *Task*

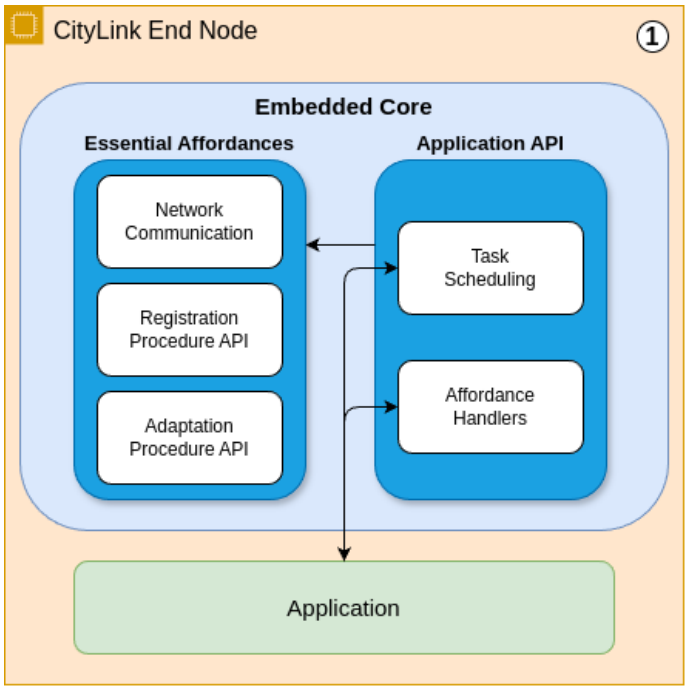


Figure 6.3: Detailed CityLink End Node software architecture (repeated)

Scheduling API and a set of *Affordance Handlers* used by application code to implement the *Interaction Affordances* defined in its *Application TM*.

- **Application Module**: The application code is loaded at runtime, using the *Embedded Core* API to interact with the end node hardware and the IoT system.

Booting from a cold start (the device was powered off), the MicroPython firmware is loaded from the end node's flash memory, initializing any hardware drivers and setting up the MicroPython interpreter.

```
1  from citylink.core import main
2
3  try:
4      main()
5  except Exception as e:
6      print(f"[BOOT] CityLink EmbeddedCore failed with error: {e}")
7
8  from machine import soft_reset
9
10 print("[BOOT] Resetting the device...")
11 soft_reset()
```

Listing 6.1: uMQTT Core Boot File

Once the interpreter is ready, it searches for the *boot.py* file. This file is automatically executed by the MicroPython interpreter, initializing the **any** *Embedded Core* runtime on the device, regardless of the protocol it implements. This means that the boot file is not specific to the *uMQTT Core* implementation, but rather a generic entry point for any MicroPython-based *Embedded Core* implementation that may be present on the device. Listing 6.1 presents the *boot.py* file used in the proof-of-concept.

The *boot.py* file imports the *main* function from the *citylink.core* module, which is the main entry point for the *uMQTT Core* implementation. Once again, the *citylink.core* module is not specific to the proof-of-concept implementation, but rather the expected entry point for any runtime in the family. Once installed on an end-node, any MicroPython-based *Embedded Core* implementation is expected to present the following directory structure:

```
/ <- File System Root
|--- boot.py <- MicroPython boot file
|--- main.py <- application entry point
|--- citylink/
|------ core.py <- Embedded Core main module
|------ ext/ <- Extension modules directory
|--------- ext1.py
|--------- ext2.py
|--------- ...
|--- config/
|------ config.json <- Default configuration file
|------ ...
|--- ...
```

Where

- *boot.py* is the MicroPython boot file, with the content presented in listing 6.1;
- *main.py* is the application entry point, invoked by once the *Embedded Core* runtime is initialized;
- *citylink/* is the directory containing the *Embedded Core* implementation, with the *core.py* module implementing the essential affordances for registration and adaptation;
- *ext/* is the directory containing the extension modules, which implement the affordances required by applications to interact with the end node, and any other desired functionality;
- *config/* is the directory containing the configuration files for the *Embedded Core* implementation. In particular, the default config.json includes the network configuration for the end node, such as the Wi-Fi credentials and the MQTT broker address.

- Any other files may be in the file system, being loaded by any modules. In the proof-of-concept case, an MQTT client library is included, under */umqtt/simple.py* [57].

Represented in Figure 6.4 are the primary operational states of an end node running the *uMQTT Core* implementation, including the initialization step as described above.

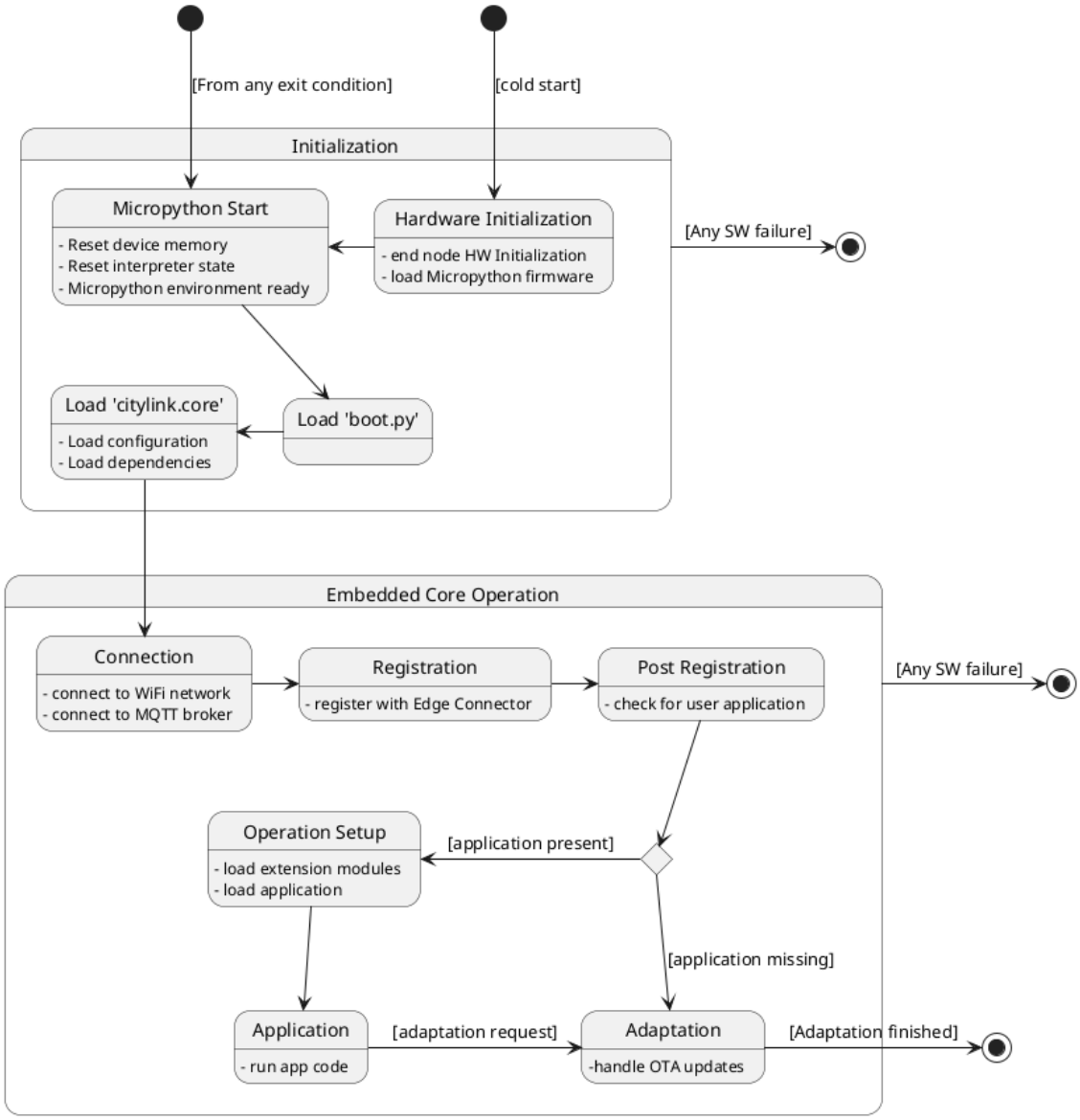


Figure 6.4: uMQTT Core Operational States

Going through the states represented in the figure, after the initialization step, the end node first attempts to connect to the Wi-Fi network and the MQTT broker using the credentials provided in the *config.json* file. After a successful connection, the end node proceeds to the registration step, identifying itself to the *Edge Connector* service. Post registration, the end node checks for the presence of an application module to load (*main.py*) and starts it if present, proceeding into the *Operation Setup* and *Application* states. The *Operation Setup* state is responsible for loading the

extension modules found in the *ext/* directory, which implement the affordances required by the application to interact with the end node, and fill in any missing functionality that is not deemed essential for the end node to operate.

This logic of loading extension modules at runtime is a key feature of the proof-of-concept implementation, allowing for a flexible and extensible architecture. It allows for the essential affordances to be implemented in a single file, reducing it to a minimum set of features required for the end node to operate. This way, as part of adaptation requests, new extension modules can be added, removed, or updated, without compromising the end node's essential affordances. Similarly, if an update of the core module is required, only a single file must be changed to ensure minimum operational capabilities.

Continuing with the Figure 6.4 analysis, the end node remains in the *Application* state until it receives an adaptation request from the *Edge Connector* service, transitioning to the *Adaptation* state. This state can also be entered if, post registration, no application module is found. In this case, the end node published its status as waiting for updates. It remains in the *Adaptation* state until the adaptation is completed, issuing a software reset to the MicroPython firmware to apply the changes. This reset returns the node to the *Initialization* state, where it will reinitialize the *Embedded Core* runtime and attempt to connect to the network and the MQTT broker again. Afterwards, the node will re-register with the *Edge Connector* service, waiting for the end node to return online after the reset.

Any software failure during initialization or operation of the end node will result in a reset of the MicroPython firmware, returning the node to the *Initialization* state in an attempt to recover safely. It is up to the edge node to monitor the end node's status and take appropriate actions if it detects that the end node keeps resetting itself, such as notifying the consumer or attempting to adapt the end node to a more stable state. Depending on the type of failure, the end node may forcefully go into the *Adaptation* state, where it will signal the edge node that it needs to be adapted.

6.2.1 Application API

The *uMQTT Core* runtime provides a lightweight API that allows developers to implement application logic directly from an *Application Thing Model*. This API maps closely to the Web of Things (WoT) interaction affordances—properties, actions, and events—making it straightforward to turn a semantic specification into functional MicroPython code.

At the heart of each application is a `setup()` function, which acts as the entry point and is expected to be defined in the *main.py* file. The runtime invokes this function during initialization and is responsible for registering all affordances the application provides. For example, a temperature-sensing application may declare a WoT property called "temperature":

```
core.create_property("temperature", 0.0)
```

Listing 6.2: uMQTT Core `create_property` affordance handler

This makes the property observable and readable by external consumers. When the application updates the property—e.g., after sampling a sensor—it uses:

```
core.set_property("temperature", temperature)
```

Listing 6.3: uMQTT Core `set_property` affordance handler

Actions are exposed similarly. Developers register action handlers using:

```
core.register_action_executor("toggleAlarm",
 ↪  toggle_alarm_action)
```

Listing 6.4: uMQTT Core `register_action_executor` affordance handler

This maps to the corresponding action affordance defined in the TM. The handler can be synchronous or asynchronous, depending on the function definition.

The runtime also supports event emission:

```
core.emit_event("overheating", event_data)
```

Listing 6.5: uMQTT Core `emit_event` affordance handler

This allows the application to notify observers when specific conditions are met, such as exceeding a temperature threshold.

While the API also provides utilities for task scheduling and other general quality-of-life functions for developers, these are auxiliary to the main WoT interaction pattern. They are used, for example, to create periodic tasks that sample sensors or check application state.

In addition, *Application TM* snippets can be matched to the application code above. The *temperature* property can be defined as a read-only property, as it is only updated by the end node and not by consumers (Listing 6.6). The *toggleAlarm* action is defined as an action that takes a boolean input parameter, allowing consumers to toggle the alarm state (Listing 6.7). Finally, the *overheating* event is defined to be emitted when the temperature exceeds the predefined threshold, with a data structure containing the timestamp, alarm state, and temperature value (Listing 6.8). A complete example of an *Application TM* for the application is presented in Listing A.4 in Appendix A,

and a complete API reference is available in the *CityLink Repository* [52] and in Appendix B. An application code example is also provided in Appendix C, Listing C.1. It demonstrates how to compose a functional but straightforward smart device using only the *Interaction Affordance*-based API and a small amount of code.

Listing 6.6: Temperature property declaration

```
"properties": {
    "temperature": {
        "description":
         ↪ "Sampled value in
         ↪ Celsius",
        "type": "integer",
        "readOnly": true,
        "writeOnly": false
    }
}
```

Listing 6.7: Toggle Alarm action declaration

```
"actions": {
    "toggleAlarm": {
        "description": "Toggle
         ↪ alarm. True/False
         ↪ - On/Off",
        "input": { "type":
         ↪ "boolean" }
    }
}
```

Listing 6.8: Overheating event declaration

```
"events": {
    "overheating": {
        "title": "Overheating event",
        "description": "Temperature exceeded 40C threshold",
        "data": {
            "type": "object",
            "properties": {
                "timestamp": {
                    "type": "string",
                    "format": "date-time",
                },
                "alarm": { "type": "boolean" },
                "temperature": { "type": "integer" }
            }
        }
    }
}
```

6.2.2 Adaptation API

The *CityLink* adaptation mechanism allows edge nodes to modify the configuration or behavior of end nodes by remotely updating their virtual file system over MQTT.

6.2.2.1 Overview

The adaptation process operates from a minimal runtime environmentnt and is based on a patch-style strategy: instead of re-deploying the entire end node, the edge node computes the difference between the current and target filesystem states. It transfers only the necessary file updates or deletions.

Figure 6.5 illustrates the relevant message exchanges between the edge node and the end node during adaptation.

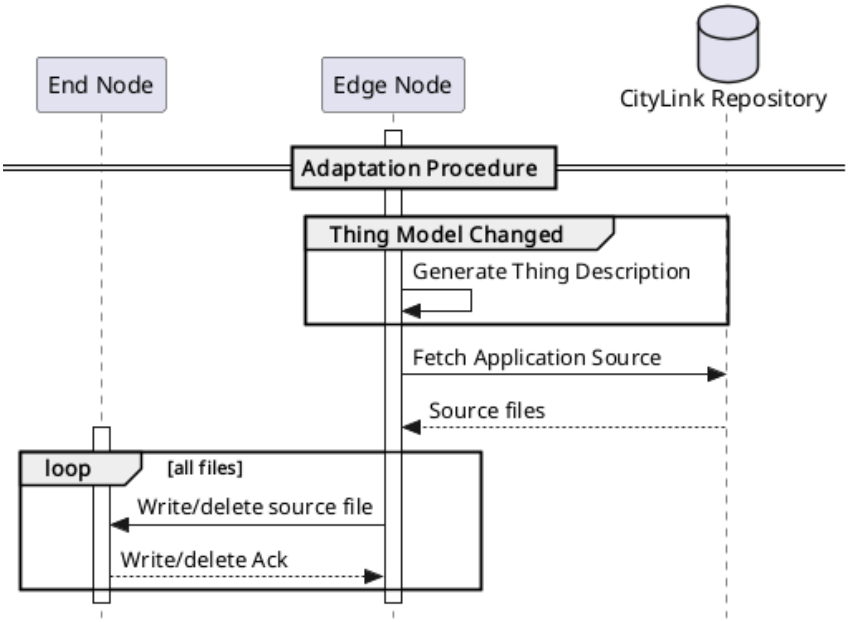


Figure 6.5: CityLink End Node Adaptation Detail

All communication is carried out using the MQTT protocol, with topics scoped to the individual end node using a prefix of the form `citylink/{node_id}/`. For clarity, the broker (e.g., Eclipse Mosquitto [55]) is omitted from the diagrams.

6.2.2.2 Interaction Affordance-Based File Management

The adaptation API comprises four *Action Affordances* and one *Event Affordance*. These are defined in the *uMQTT Core TM* and implemented by the runtime.

- **OTAUInit** – Signals the end node to reboot into adaptation mode.
- **OTAUWrite** – Transfers a new file or appends to an existing one on the virtual file system.

- **OTAUDelete** – Requests deletion of an existing file or directory path.
- **OTAUFinish** – Commits or discards the adaptation and triggers reboot.
- **OTAUResult** – Event emitted to report the outcome of file operations.

These affordances follow standard WoT conventions and are exposed on MQTT topics like `actions/core/otau/write` and `events/core/otau/report`. The full specification of payload formats, field names, and topic structure is provided in Appendix D.

6.2.2.3 Adaptation Sequence

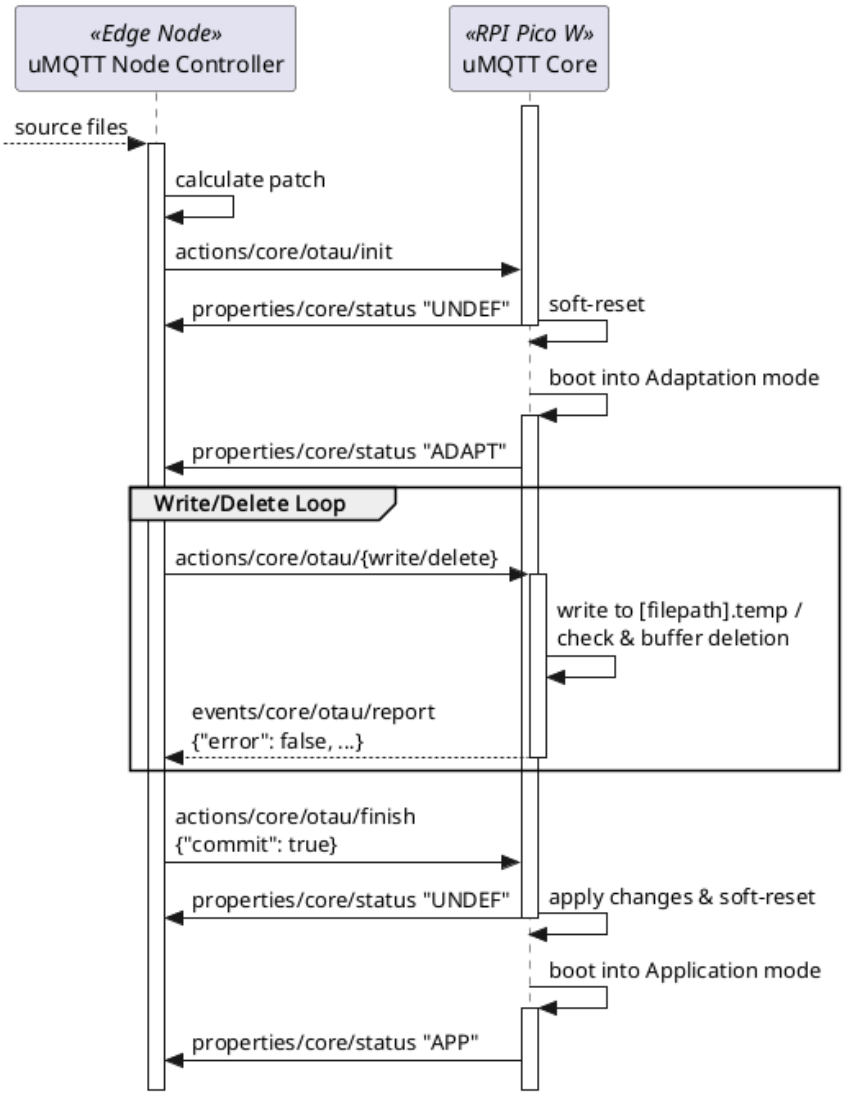


Figure 6.6: uMQTT Core Adaptation Sequence Diagram

1. Patch Calculation. The edge node begins by calculating the patch that transforms the current state of the end node's file system into the target version. Files present in the new version but not in the current one are marked for writing, while missing files are marked for deletion. Files with updated contents are flagged for overwrite.

2. Entering Adaptation Mode. The edge node invokes the 'OTAUInit' action to begin the update. This causes the end node to enter the *Adaptation* state by rebooting into a minimal runtime environment. Before rebooting, it publishes a property update on `properties/core/status` to indicate that it is temporarily entering an undefined state.

3. File Transfer. Once rebooted, the end node emits a new status property indicating readiness to receive adaptation instructions. The edge node then sends a sequence of 'OTAUWrite' and 'OTAUDelete' requests. File writes are buffered with a temporary '.temp' suffix and verified using integrity hashes. Delete requests are buffered for later processing.

4. Committing or Rolling Back. After all updates have been staged, the edge node sends an 'OTAUFinish' action with a 'commit' flag. If 'true', the end node applies the patch: temporary files are renamed (overwriting any existing ones), and deletion buffers are processed. If 'false', all staged changes are discarded, and the end node reboots into the previous configuration.

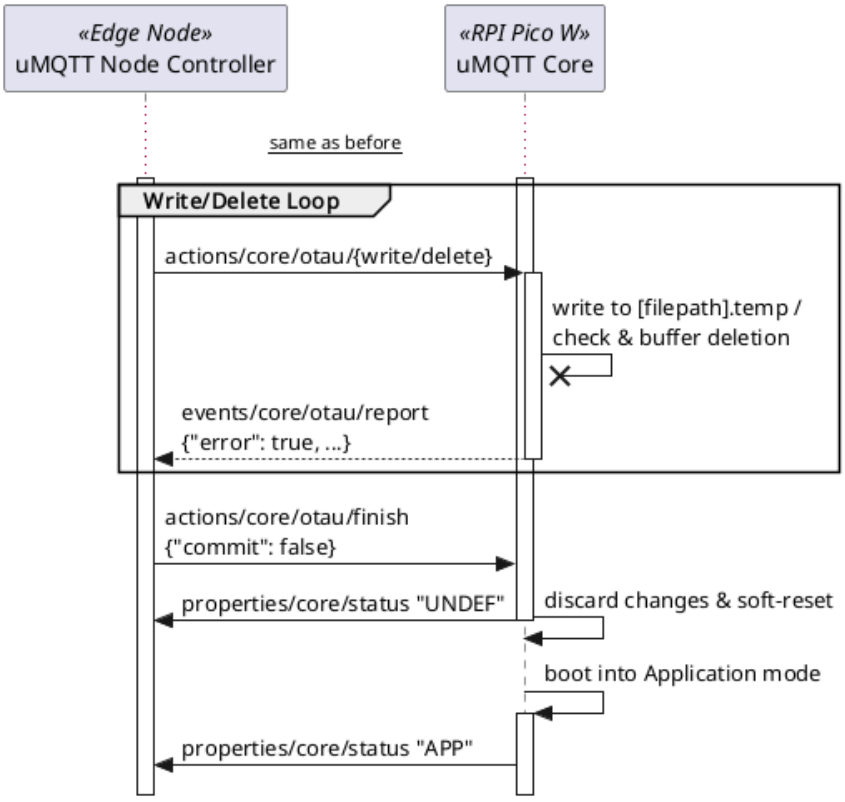


Figure 6.7: uMQTT Core Adaptation Sequence Diagram with Failure

5. Reboot and Recovery. Following a commit or rollback, the end node reboots and reloads the *Embedded Core* runtime. It re-registers with the edge node, resuming normal operation if the adaptation succeeded. If the previous configuration was faulty, it remains in adaptation mode and awaits further updates.

Figure 6.6 shows the complete adaptation sequence, and Figure 6.7 shows a rollback scenario due to an error.

6.2.2.4 Failure Handling and Rollback

If an error occurs during the adaptation (e.g., due to a file hash mismatch or interrupted transfer), the edge node may cancel the update using ‘OTAUFinish‘ with ‘commit: false‘. The end node will then discard all temporary files and delete the buffered deletions, effectively reverting to its last known good state.

In the rare case where the previous configuration is also invalid, the node remains in adaptation mode, awaiting a valid update.

6.2.2.5 Design Considerations

This adaptation approach is designed to be robust and fault-tolerant:

- All file changes are staged using ‘.temp‘ suffixes to avoid accidental overwrites.
- *Adaptation Affordances* are exposed in a minimal runtime environment, ensuring recovery is always possible.
- The core module of the *uMQTT Core* implementation that exposes the *Adaptation Affordances* can be atomically updated or replaced.
- Each file operation’s result is reported via the ‘OTAUResult‘ event, allowing the edge node to detect and recover from failures.

This design minimizes the risk of leaving end nodes in an undefined or irrecoverable state, ensuring that adaptation failures can always be retried or rolled back safely.

6.3 Edge Node Implementation

As mentioned in the introductory section of this chapter, a set of edge node components was implemented to support the proof-of-concept deployment. These components include an *Edge Connector* implementation for the MQTT protocol coupled with a *Node Controller* module developed to match the *uMQTT Core* implementation. Additionally, a *Thing Description Directory* providing the *Things, Search and Events APIs* was implemented to support the discovery of end nodes and their affordances, as well as the management of TMs and applications via the *CityLink*-specific APIs. Finally, a minimal web interface was developed, tying all the components together in a single web application for a better user experience and the proof-of-concept demonstration.

These components (aside from the web interface) are implemented as a collection of TypeScript libraries, using the Deno 2.0 runtime [58], the MQTT.js library [59]. Some additional libraries from the Eclipse ThingWeb project [49] to facilitate the manipulation of TMs and *Thing Descriptions*.

Sharing the **@citylink-edgenode** namespace, the developed libraries are available in the *CityLink Repository* on GitHub [52], with the main components being:

- **@citylink-edgenode/core**: The main library defining the interfaces and base classes for the *Edge Connector* and *Node Controller* implementations, together with common utilities and types used by the other libraries to work with the established *CityLink* data model.
- **@citylink-edgenode/connector-mqtt**: The implementation of the *Edge Connector* service for the MQTT protocol.
- **@citylink-edgenode/controller-umqtt-core**: The implementation of the *Node Controller* for the *uMQTT Core* implementation, providing the necessary methods to manage adaptation requests targeting end nodes running the *uMQTT Core* implementation.
- **@citylink-edgenode/thing-directory**: The implementation of the *Thing Description Directory* component, providing the *Things, Search and Events APIs* as well as the *CityLink*-specific APIs for end node adaptation and thing model management.

The web interface is implemented as a separate Deno application, using the Deno Fresh library [60] to provide a simple and responsive user interface for interacting with the *Edge Connector* service and the TDD.

6.3.1 Edge Connector Components

The *Edge Connector* service is implemented in TypeScript, using the Deno 2.0 runtime [58] and the npm MQTT library [59]. Structurally, a working *Edge Connector* implementation is composed of two elements, as previously established: a module in charge of listening for registration requests from end nodes and one or more modules in charge of managing adaptation requests targeting end nodes.

As such, the class hierarchy presented in Figure 6.8 was established as the basis for the **@citylink-edgenode** libraries.

A central abstract class defines the core logic of the *Edge Connector* component. This class establishes a common interface and shared functionality for all concrete *Edge Connector* implementations, regardless of the underlying communication protocol. Each implementation must define its protocol-specific behavior for handling registration requests from end nodes.

The *Edge Connector* maintains a registry of controller factories, modular components responsible for managing compatible end nodes. These factories advertise compatibility through a descriptor that includes the title and version of *the Node Controller TM* they support.

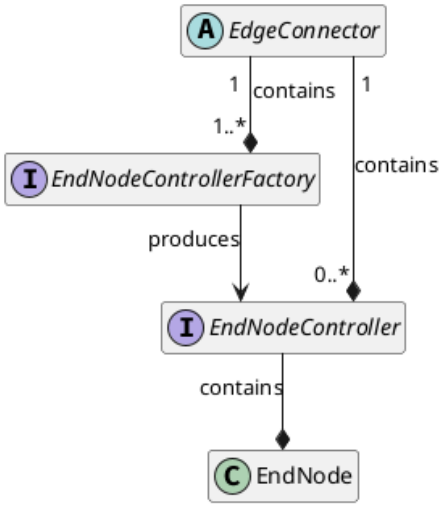


Figure 6.8: **@citylink-edgenode/core** class diagram

As described in the *Relational TM Schema* (Section 4.5), an *Embedded Core TM* may declare a `citylink:controlledBy` relationship linking it to a compatible *Node Controller TM*. When a new end node registers, the *Edge Connector* retrieves this linked *Node Controller TM* and compares its metadata to the compatibility descriptors of the registered factories. If a matching factory is found, the *Edge Connector* uses it to instantiate a new controller for the end node.

The instantiated controller is responsible for managing the lifecycle and behavior of the end node. This includes starting or stopping control processes and initiating adaptations based on TM updates or runtime conditions. Applying this class hierarchy to the proof-of-concept implementation, Figure 6.9 presents the complete class diagram for all the **@citylink-edgenode** libraries.

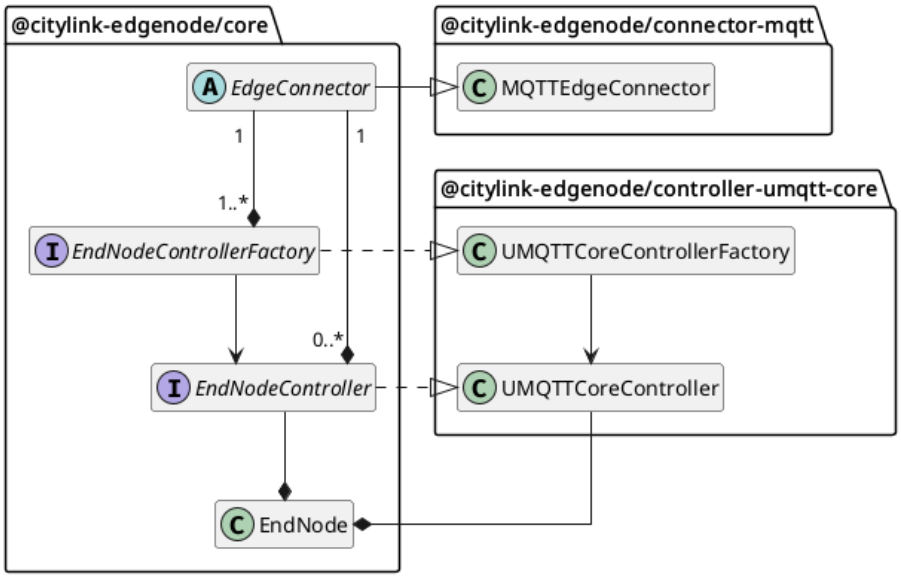


Figure 6.9: **@citylink-edgenode/core** class diagram extended

6.3.2 Registration API

Contrary to the application and adaptation APIs presented in Sections 6.2.1 and 6.2.2, respectively, the registration API is not defined as a set of affordances in the *uMQTT Core* implementation, but rather imposed by the *Edge Connector* service.

Once again simulating client-server interactions over MQTT, the registration API is defined in the *Edge Connector TM* for the `MQTTEdgeConnector` service, as a pair of action and event affordances. Listings 6.9 and 6.10 present snippets of the TM with the relevant affordances.

```
"title": "End Node Registration",
"description": "Register a new end node with the connector.",
"input": {
    "type": "object",
    "description": "Input for registering an end node with the
     ↪ connector.",
    "properties": {
        "tm": {
            "type": "string",
            "description": "URL of the Thing Model describing the
             ↪ end node."
        },
        "placeholder": {
            "type": "object",
            "description": "Key-value pairs for additional
             ↪ parameters required for registration.",
        }
    },
    "required": [ "tm" ]
},
"forms": [
    {
        "href": "{{CITYLINK_HREF}}",
        "mqv:topic": "citylink/{nodeID}/registration",
        "contentType": "application/json",
        "op": "invokeaction",
        "mqv:qos": 2,
        "mqv:retain": false
    }
]
```

Listing 6.9: End Node registration action contents (actions/registration)

For brevity, the concrete data types for the `response` event are omitted from Listing 6.10, as well as the definition of the `{nodeID}` URI variable, which represents a unique ID of the end node, **before** registration. This ID keeps MQTT topics unique for each end node, allowing the *Edge Connector* service to manage multiple end nodes concurrently. After registration, end

```
"title": "End Node Registration Response",
"description": "Event triggered when an end node registration is
 ↪  acknowledged by the controller.",
"data": {
    "oneOf": [
        {/* Ack response */},
        {/* Error response */},
        {/* Success response */}
    ]
},
"forms": [
    {
        "href": "{{CITYLINK_HREF}}",
        "mqv:topic": "citylink/{nodeID}/registration/ack",
        "contentType": "application/json",
        "op": [ "subscribeevent", "unsubscribeevent" ],
        "mqv:qos": 2,
        "mqv:retain": false
    }
]
```

Listing 6.10: End Node registration response event contents (events/response)

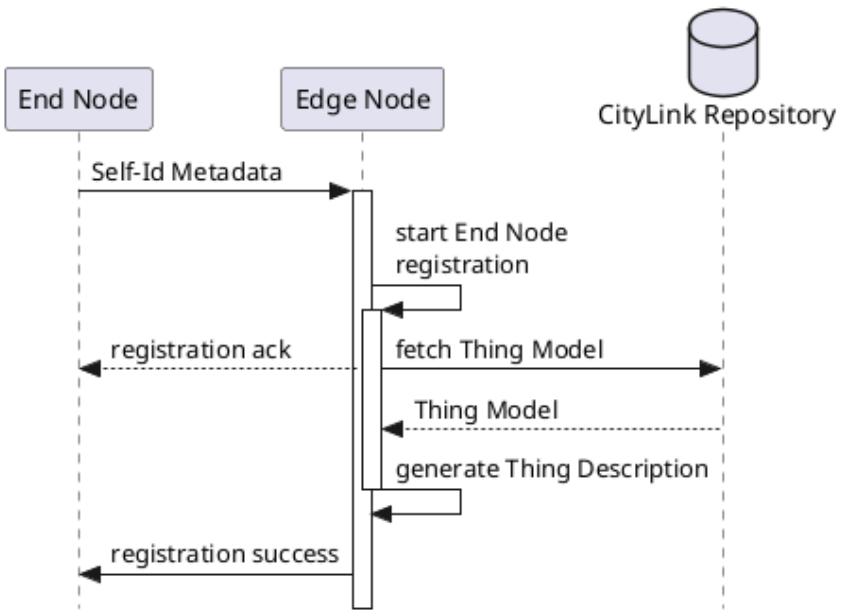


Figure 6.10: End Node Registration Sequence Diagram (repeated)

nodes use ID values attributed by the *Edge Connector* service, generally version 4 UUIDs [22] as previously mentioned. Before registration, however, end nodes use a temporary ID generated by the *Embedded Core* runtime, which is used to identify the end node solely during the registration process. Additionally, the `{{CITYLINK_HREF}}` placeholder is used to dynamically assign the `href` property to the actual URL of the MQTT Broker, once the TM is processed into a *Thing Description* before being served to consumers.

Regarding the sequence of events for the registration process itself, the implementation is identical to the conceptual sequence presented in Chapter 4, Figure 4.4 (repeated here in Figure 6.10).

The complete definitions for both registration *Interactions Affordances* can be consulted in Appendix A, Listings A.6 and A.7, respectively.

6.4 Thing Description Directory

Per the *W3C Web of Things Discovery Standard* [32], the *Thing Description Directory* (TDD) is a *Thing* that provides a service to manage a set of *Thing Descriptions* describing other *Things*. A WoT TDD can provide a set of APIs to manage the *Thing Descriptions* it serves. These APIs include:

Things API. A RESTful HTTP API served at the `/things` endpoint, providing interfaces to execute CRUDL operations on *Thing Descriptions*.

Events API. An optional notification API based on the *Server-Sent Events* protocol, served at the `/events` endpoint, allowing consumers to subscribe to events attributed to the lifecycle of *Thing Descriptions* in the Directory, including the `thing_created`, `thing_updated`, and `thing_deleted` event types.

Search API. An optional search API served at the `/search` endpoint, allowing consumers to search for *Thing Descriptions* based on their properties and affordances. Three search APIs can be supported:

- *Syntactic Search using **JSONPath***: A search API served at the
  `/search/jsonpath?query={query}`
  endpoint, based on the JSONPath query language [61].
- *Syntactic Search using **XPath***: A search API served at the
  `/search/xpath?query={query}`
  endpoint, based on the XPath query language [62].
- *Semantic Search using **SPARQL***: A search API served at the
  `/search/sparql`
  endpoint, based on the SPARQL query language [26].

Syntactic search APIs allow consumers to filter *Thing Descriptions* based on key-value pairs. In contrast, semantic search APIs enable filtering based on semantic relationships between data, leveraging the JSON-LD linked data paradigm.

CityLink TDD Extensions. Within the *CityLink* framework, the implementation of the *Thing Description Directory* may fully support the *Search* and *Events* APIs while limiting the *Things API* to read-only operations—allowing consumers to list and retrieve *Thing Descriptions*, but not create, update, or delete them.

In addition, *CityLink* extends the standard WoT TDD specification with a set of framework-specific APIs that enable end node adaptation, TM management, and application lifecycle control. These extensions are provided as part of the *@citylink-edgenode/thing-directory* library, which supports:

- `list` and `retrieve` operations on *Thing Descriptions*,
- All three event types from the Events API (`thing_created`, `thing_updated`, and `thing_deleted`),
- JSONPath-based syntactic search.

Beyond these baseline features, the following *CityLink*-specific APIs are exposed:

- **Adaptation API** (`/adaptation`): Allows clients to initiate end node adaptation via HTTP POST, accepting either an *Application TM* or a URL pointing to one. This API is essential to the dynamic update mechanism central to the framework.
- **Thing Model API** (`/thing-models`): Provides CRUDL operations for managing TMs via standard HTTP verbs (POST, GET, PUT, DELETE). This supports the versioning and reuse of reusable hardware/software abstractions.
- **Application API** (`/applications`): Enables management of deployed applications, also using standard REST operations. Applications in this context are structured as *CityLink Application TMs*, linking software deployments to abstract device descriptions.

Although secondary to the WoT-defined APIs, these *CityLink*-specific services, particularly the **Adaptation API**, are critical to the operation of the overall framework. Application deployment and runtime adaptation of end nodes depend on these extensions. In turn, managing this process necessitates supporting infrastructure for discovering and maintaining TMs and application definitions.

In practical deployments, the standard WoT TDD APIs would primarily serve consumers such as smart city residents, developers, or third-party applications, enabling them to discover and interact with available devices and services via their *Thing Descriptions*. Conversely, the *CityLink*-specific APIs are geared toward privileged consumers such as city administrators and automated infrastructure controllers, who manage the underlying network of end nodes—deploying applications, adapting configurations, and ensuring long-term operation of the Smart City fabric.

6.5 Proof-of-Concept Deployment

The proof-of-concept deployment, as introduced at the beginning of this chapter, was realized using an x86_64 Linux machine as the edge node and a Raspberry Pi Pico W [56] as the end node. To provide context, Figure 6.11, repeated here for reference, illustrates the overall deployment architecture, showing the edge and end node components alongside the MQTT broker responsible for message exchange.

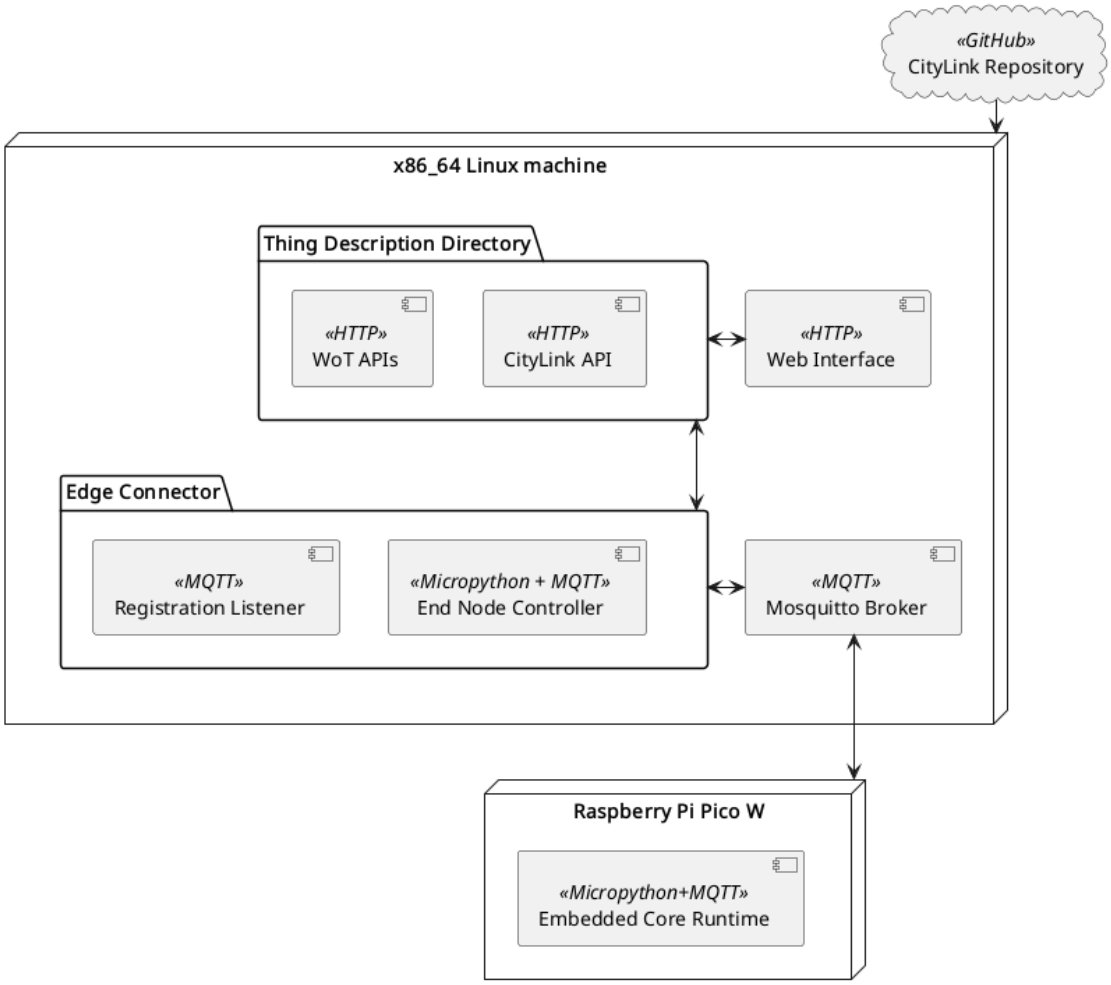


Figure 6.11: CityLink Proof-of-Concept Deployment (repeated)

6.5.1 Edge Node Setup

On the edge node, the deployment leverages the **@citylink-edgenode** suite of libraries to instantiate and launch an *Edge Connector* service for the MQTT protocol. The service integrates several key features described throughout this chapter and uses a minimal TypeScript script to launch each component.

The *Edge Connector* is configured using a *Edge Connector TM*, which is processed into a concrete *Thing Description* at runtime, which, in turn, provides the configuration and semantic metadata necessary for the connector's operation.

In addition to the connector itself, a *Node Controller Factory* is registered to handle *uMQTT Core* end nodes. This factory is initialized with a *Node Controller TM*, which it uses to generate *Thing Descriptions* for individual controller instances as end nodes register.

Finally, the connector instance is registered with the *Thing Description Directory* service running locally on the edge node. This enables dynamic discovery and management of *Thing Descriptions* for connected end nodes, completing the *CityLink* runtime stack for this proof-of-concept scenario.

A complete code snippet can be found in Appendix C, Listing C.2.

6.5.2 Web Interface

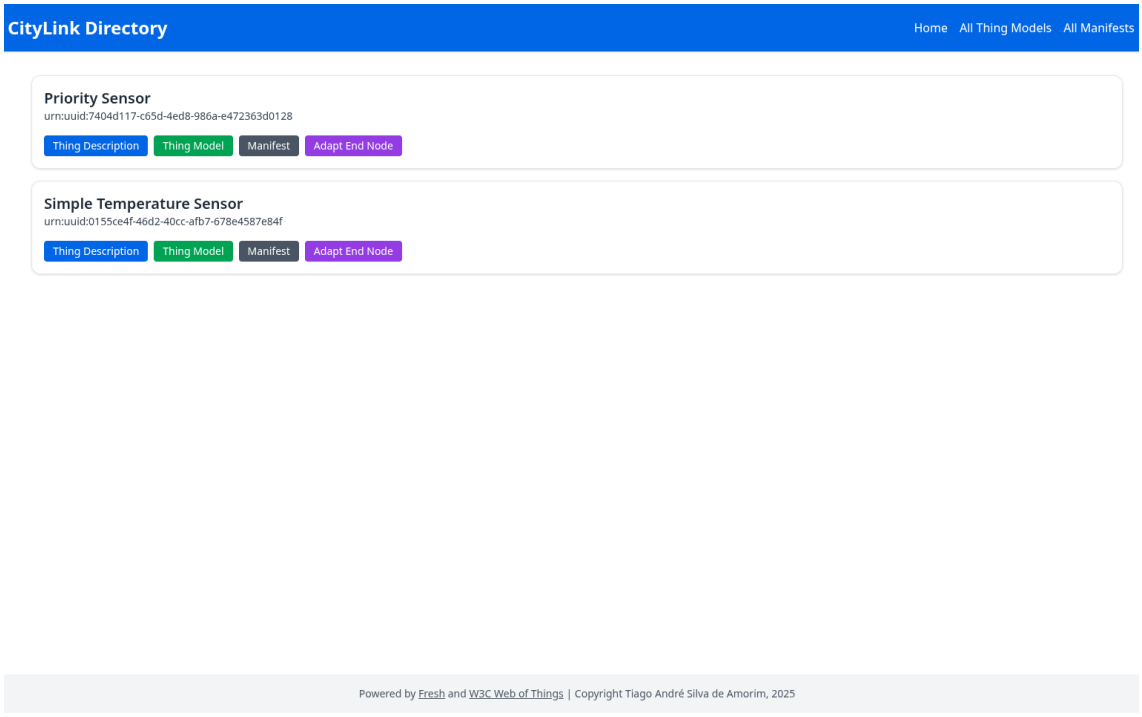


Figure 6.12: CityLink Web Interface Screenshot

To complement the TDD APIs, a prototype web interface was developed using the Deno Fresh framework [60]. This standalone UI consumes the TDD's public API and provides a user-friendly frontend to view and interact with registered devices.

Built with Fresh, a web framework optimized for edge performance based on the lightweight PreactJS library [63], the interface is well-suited for minimal deployments. Figure 6.12 presents a screenshot of the interface homepage, displaying two registered end nodes: a *Priority Sensor* and a *Simple Temperature Sensor*. A UUID under the `urn:uuid:` namespace [22] identifies each of the end nodes.

These devices correspond to two Raspberry Pi Pico W boards running the *uMQTT Core* variant of the *Embedded Core* runtime. The *Simple Temperature Sensor* executes an example application introduced earlier, while the *Priority Sensor*'s application is explored in the following sections.

6.5.3 Interaction Affordance Overview

Clicking the *Thing Description* button on the UI reveals interaction affordances for the *Priority Sensor*, shown in Figure 6.13.


```
{ 10 items
  "@context" : [ 2 items
    0 : string "https://www.w3.org/2022/wot/td/v1.1"
    1 : string "https://raw.githubusercontent.com/les2feup/CityLink/refs/heads/main/context.jsonld"
  ]
  "@type" : [ 2 items
    0 : string "Thing"
    1 : string "citylink:AppTM"
  ]
  "title" : string "Priority Sensor"
  "description" : string "A simple sensor that publishes readings on different topics based on a priority level."
  "links" : [...] 1 item
  "properties" : { 11 items
    "priority" : {...} 8 items
    "citylink:platform_MCU_cores" : {...} 8 items
    "citylink:platform_MCU_SRAM" : {...} 8 items
    "citylink:platform_MCU_flash" : {...} 8 items
    "citylink:platform_MCU_DMA" : {...} 8 items
    "citylink:platform_MCU_USB" : {...} 8 items
    "citylink:platform_MCU_peripherals" : {...} 8 items
    "citylink:platform_NIC_wifi" : {...} 7 items
    "citylink:platform_NIC_bluetooth" : {...} 7 items
    "citylink:embeddedCore_logLevel" : {...} 8 items
    "citylink:embeddedCore_status" : {...} 8 items
  }
  "events" : { 3 items
    "sensor_value" : {...} 7 items
    "citylink:embeddedCore_log" : {...} 4 items
    "citylink:embeddedCore_OTAUReport" : {...} 4 items
  }
  "actions" : { 4 items
    "citylink:embeddedCore_OTAUInit" : {...} 5 items
    "citylink:embeddedCore_OTAUFinish" : {...} 5 items
    "citylink:embeddedCore_OTAUWrite" : {...} 6 items
    "citylink:embeddedCore_OTAUDelete" : {...} 6 items
  }
  "forms" : [...] 2 items
  "id" : string "urn:uuid:b0d16250-7fed-4ed3-a840-00f960558a45"
}
```


Figure 6.13: CityLink Web Interface Thing Description Screenshot

The figure illustrates three affordance naming schemas:

- **No prefix:** Application-specific affordances, such as `priority` or `sensor_value`, are defined in the *Application TM* and copied directly to the *Thing Description*.

- **citylink:embeddedCore:** Runtime-level affordances (e.g., `status`, `OTAU*`) that originate from the *Embedded Core TM*.

- **citylink:platform:** Platform-specific affordances (e.g., CPU model, memory) extracted from the *Platform TM*.

The edge connector automatically applies these prefixes at registration time, simplifying application development and ensuring namespace separation. This same philosophy is also applied to the naming schemas used for MQTT topics, which are subject to pre-processing *Thing Descriptions* are generated from the TMs. Affordance topics are structured using the affordance types (property, action, event) and the `core` and `app` namespace tags. This MQTT topic structure was designed with the standard MQTT wildcards in mind, allowing consumers to subscribe to patterns such as

```
citylink/+/properties/core/#
```

or

```
citylink/+/events/app/#
```

Appending E contains additional screenshots capturing the other pages of the Web Interface. Figure E.4 shows a detailed view of the `priority` property, which lets the consumer influence the publishing topic of the end node. Valid values include `"low"`, `"medium"`, and `"high"`. This dynamic adjustment enables simple yet effective prioritization strategies in larger deployments.

Figure E.5, which presents the `sensor_value` event affordance in detail. This event contains two `forms` entries, allowing consumers to choose the subscription topic for this event. The first entry defines the subscriptions to a single priority level, using the `subscribeevent` operation on a topic ending with the value of the `priority` URI variable. Conversely, consumers may simultaneously subscribe to all priority levels using the catch-all topic in the second `forms` entry. The data received when subscribing to one of the event topics will be an integer value, between 0 and 100, representing the sensor reading.

Additionally, Figures E.2 and E.3 present the user interface for the *Adaptation* endpoint. For this, the UI provides an interface where the ID and TM title for the end node undergoing adaptation are presented, an input field where the URL for the new TM can be inserted, and a TM preview window, allowing the user to check the new TM before initiating the adaptation procedure.

6.5.4 Runtime Logging and Diagnostics

```
[2025-06-15 19:20:12.435 +0100] INFO: CityLink:controller:UMQTTCore 📩 New message received
    node: "de4c0444-c06b-4c3b-85a9-493fa6c87814"
    topic: "citylink/de4c0444-c06b-4c3b-85a9-493fa6c87814/properties/app/temperature"
    type: "properties"
    namespace: "app"
    affordance: "temperature"
    message: "16.31189"
[2025-06-15 19:20:13.325 +0100] INFO: CityLink:controller:UMQTTCore 📩 New message received
    node: "b0d16250-7fed-4ed3-a840-00f960558a45"
    topic: "citylink/b0d16250-7fed-4ed3-a840-00f960558a45/events/app/sensor_value/low_prio"
    type: "events"
    namespace: "app"
    affordance: "sensor_value/low_prio"
    message: "23"
[2025-06-15 19:20:13.436 +0100] INFO: CityLink:controller:UMQTTCore 📩 New message received
    node: "de4c0444-c06b-4c3b-85a9-493fa6c87814"
    topic: "citylink/de4c0444-c06b-4c3b-85a9-493fa6c87814/properties/app/temperature"
    type: "properties"
    namespace: "app"
    affordance: "temperature"
    message: "16.70253"
[2025-06-15 19:20:14.325 +0100] INFO: CityLink:controller:UMQTTCore 📩 New message received
    node: "b0d16250-7fed-4ed3-a840-00f960558a45"
    topic: "citylink/b0d16250-7fed-4ed3-a840-00f960558a45/events/app/sensor_value/low_prio"
    type: "events"
    namespace: "app"
    affordance: "sensor_value/low_prio"
    message: "77"
```

Figure 6.14: Edge Connector Logs - Part 1

```
[2025-06-15 19:21:19.979 +0100] INFO: CityLink:controller:UMQTTCore 📩 New message received
    node: "b0d16250-7fed-4ed3-a840-00f960558a45"
    topic: "citylink/b0d16250-7fed-4ed3-a840-00f960558a45/properties/app/priority"
    type: "properties"
    namespace: "app"
    affordance: "priority"
    message: "\"high\""
[2025-06-15 19:21:20.369 +0100] INFO: CityLink:controller:UMQTTCore 📩 New message received
    node: "b0d16250-7fed-4ed3-a840-00f960558a45"
    topic: "citylink/b0d16250-7fed-4ed3-a840-00f960558a45/events/app/sensor_value/high_prio"
    type: "events"
    namespace: "app"
    affordance: "sensor_value/high_prio"
    message: "62"
[2025-06-15 19:21:20.483 +0100] INFO: CityLink:controller:UMQTTCore 📩 New message received
    node: "de4c0444-c06b-4c3b-85a9-493fa6c87814"
    topic: "citylink/de4c0444-c06b-4c3b-85a9-493fa6c87814/properties/app/temperature"
    type: "properties"
    namespace: "app"
    affordance: "temperature"
    message: "17.82559"
```

Figure 6.15: Edge Connector Logs - Part 2

Figures 6.14 and 6.15 show logs from the edge node, representing MQTT messages being processed. These logs identify end nodes by ID and categorize messages by affordance type and namespace (*core* or *app*), showing the data being streamed from the end nodes to the MQTT broker and then to the edge node acting as a consumer.

Circling back to the analysis of the *Priority Sensor* end node, the logs in Figure 6.15 show the result of the priority property being written with the value `"high"`. Issuing a publish packet to the topic listed with the `writeproperty` operation in Figure E.4 with the value `"high"` results in the end node changing the topic to which it publishes its readings, as shown in the log message. The first log message shows the end node publishing the change to the `priority` property due to the write operation. In contrast, the second log message shows the end node publishing its first reading after the change, which is now published to the `/sensor_value/high_prio` topic.

The remaining pages and features of the web interface are not presented in detail here for brevity, being available in the *CityLink Repository* [52]. They follow a similar pattern of hooking into the backend API provided by the *Thing Description Directory* service, allowing for a more human-friendly way to interact with the services offered by the *CityLink* framework.

6.5.5 System Resource Footprint

Table 6.1 lists the source files for the *uMQTT Core* runtime, presented by name and size after being compiled to bytecode. Using the mpy-cross utility [64], the *uMQTT Core* source files were pre-compiled to MicroPython bytecode and transferred to the end nodes as part of the configuration setup before deployment. Using pre-compiled bytecode files allows for better performance and reduced memory usage, as the MicroPython runtime can skip the translation step from Python code to bytecode at runtime, which is particularly important for resource-constrained devices such as the Raspberry Pi Pico W. The only exception is the `boot.py` file, which the MicroPython runtime expects to be a regular Python script. In any case, this file is small enough that the performance impact of not pre-compiling it is negligible.

Table 6.1: File names and sizes for uMQTT Core files after being compiled to bytecode

name	size
boot.py	223 B
config/config.json	323 B
umqtt/simple.mpy	2.2 kB
citylink/core.mpy	7.5 kB
citylink/ext/aff_handler.mpy	1.2 kB
citylink/ext/task_sched.mpy	877 B

Adding up the total size of the files in Table 6.1, the total size of the *uMQTT Core* files after being compiled to bytecode is under 12.5 kB. This is a small footprint, compared to the size of the MicroPython runtime itself, which for the *Raspberry Pi Pico W* is a 1.7 MB UF2 file [65].

Leveraging the MicroPython REPL to inspect the stack utilization further, the `os.statvfs("//")` command shows that, for a block size of 4096 bytes, there are 188 available blocks, resulting in a total of 770 kB of available space on the *Raspberry Pi Pico W* flash storage. For a

regular Raspberry Pi Pico W with 2 MB of flash storage, the total flash utilization is 61.5%, most of which is taken up by the MicroPython firmware. The total size of the *uMQTT Core* represents less than 1% of the total flash storage available on the device and only 4.2% of the leftover space from the MicroPython installation, leaving plenty of room for additional applications and libraries to be deployed on the end node.

Furthermore, each file is small enough to fit in a single MQTT message easily. MQTT messages have a theoretical maximum of 256 MB, although most clients will impose a lower limit on connection. For the tests concluded in the proof of concept deployment, all files could be transferred in a single MQTT message, with even the largest file, `citylink/core.mpy`, only amounting to 7.5 kB. The RP2040 SoC [54] built into the Raspberry Pi Pico W has 264 kB of SRAM, which is more than enough to hold each of the files in memory when performing read/write operations.

Interrupting the *Priority Sensor* application via the command line after the end node is registered and publishing values, the `micropython.mem_info()` function call yields the results in Listing 6.11. From the total of 205596 bytes available to MicroPython's Garbage Collector (GC), 57648 bytes are used and 148048 are free, totaling a memory utilization of 28% during normal operation mode.

```
1 >>> import micropython
2 >>> micropython.mem_info()
3 stack: 572 out of 7936
4 GC: total: 205696, used: 57648, free: 148048
5  No. of 1-blocks: 963, 2-blocks: 135, max blk sz: 72, max free sz:
    ↪  9240
```

Listing 6.11: End node normal operation memory snapshot

Forcing a reboot into adaptation mode via the REPL and interrupting the adaptation process before any data is received from the edge node allows for diagnostics of the memory utilized by the *uMQTT Core* when just the essential affordances are loaded.

```
1 >>> import micropython
2 >>> micropython.mem_info()
3 stack: 572 out of 7936
4 GC: total: 205696, used: 34288, free: 171408
5  No. of 1-blocks: 543, 2-blocks: 86, max blk sz: 72, max free sz:
    ↪  10700
```

Listing 6.12: End node adaptation procedure memory snapshot

Running the same command as before yields the results in Listing 6.12. The allocated memory in this mode is 34288 bytes, for a total utilization of 16.7%, a 40.5% decrease relative to the

previous value. This leaves 83.3% of all usable memory (i.e., memory not reserved for the MicroPython firmware) available for the update procedure. At the superior limit, this means that, in theory, a Raspberry PI Pico W could handle single file downloads roughly up to $171kB$, equivalent to more than 20 times the file size of the largest *uMQTT Core* source file.

While performance is not key for this proof-of-concept, these data points help cement the feasibility of the *CityLink* framework and its low footprint on constrained devices, leaving plenty of resources for deploying more complex applications.

6.6 Summary

This chapter presented the implementation of the *CityLink* framework, focusing on the *uMQTT Core* end node implementation and the edge node components developed to support the proof-of-concept deployment.

The *uMQTT Core* is a minimal implementation of the *Embedded Core* runtime, providing the necessary affordances to support the adaptation of end nodes and the interaction with the *Edge Connector* service. The adaptation process was described in detail, including the steps taken to ensure a safe and reliable adaptation process and the mechanisms used to roll back changes in case of errors. The edge node components were implemented as a set of TypeScript libraries, providing an *Edge Connector* service for the MQTT protocol, a *Node Controller* implementation for the *uMQTT Core* end nodes, and a *Thing Description Directory* service to manage *Thing Descriptions*, *Thing Models*, and applications. The chapter also presented the proof-of-concept deployment, which was conducted using an x86_64 Linux machine as the edge node and a Raspberry Pi Pico W as the end node. The web interface developed to interact with the *Edge Connector* service and the *Thing Description Directory* was also presented, providing a simple way to visualize and interact with the end nodes registered with the edge node.

The primary design goal for the proof-of-concept implementation described in this chapter is to demonstrate how the conceptual framework and data models presented in chapter 4 can be implemented in practice. While trying to be as faithful as possible to the conceptual design, the implementation is not intended to be a fully-fledged production-ready solution, but rather demonstration of the technical feasibility of the proposed framework and to serve as a well-defined bases for future development and experimentation as both the W3C Web of Things standard and the *CityLink* proposal evolve.

Chapter 7

Conclusions & Future Work

This dissertation presents a holistic solution for enhancing interoperability and adaptability at the edge layer of large-scale IoT deployments, particularly in smart city environments. Motivated by the challenges of IoT fragmentation and the necessity for flexible, adaptive urban infrastructures, the proposed solution utilizes the W3C Web of Things standard and semantic web technologies to establish a consistent, machine-readable interface for devices within a monitoring infrastructure.

The proposed solution includes software components that allow end-node hardware requirements to be exchanged interchangeably, best fitting various deployment scenarios. Another key contribution is the *Relational TM Schema*, which integrates with the software elements to enable automatic device registration, over-the-air updates, and seamless integration of heterogeneous platforms, all described through semantically rich descriptions enabled by the WoT standard.

A proof of concept demonstrates the practicality of the proposed solution using MicroPython and MQTT. It showcases automatic registration, device adaptation, and simplified development workflows through reusable application models and rich APIs. This implementation validates the feasibility of the approach with lightweight, low-power devices and open-source toolchains.

Based on the results presented in this dissertation, several promising directions for future work have been identified.

One avenue involves the formal definition and structuring of *Platform TM*. This effort could draw inspiration from the DeviceTree specification [66], which is widely used in projects such as the Linux kernel and Zephyr RTOS [51] to describe hardware platforms and support automated code generation. Translating DeviceTree-like structures into a linked data representation, or even a formal Web ontology, would enable Platform TMs to express device capabilities in a machine-readable and semantically rich format. This, in turn, would support dynamic end node adaptation and edge-side code generation within the *CityLink* framework, significantly streamlining the development of new *Embedded Core* firmware variants, end node applications, and improving integration with diverse hardware platforms.

Beyond its immediate benefits for the *CityLink* ecosystem, such a contribution holds broader significance. A standardized, ontology-based representation of hardware components could serve

as a foundational building block for semantically aware development tools, automated configuration pipelines, and intelligent orchestration systems. As such, this work could meaningfully advance the state of the art in both the Semantic Web and embedded systems domains by introducing a rich, interoperable vocabulary for hardware description that extends well beyond the smart city context.

Enhancing the adaptive capabilities of Edge Nodes is another relevant direction. Mechanisms for automatic edge node adaptation based on the presence and characteristics of nearby devices could increase network resilience and responsiveness, enabling more efficient reconfiguration and orchestration in dynamic environments.

From a data management perspective, integrating semantic search engines and linked data stores such as RDF databases with SPARQL support into the CityLink architecture offers the potential to unlock advanced features for discovery, orchestration, and contextual reasoning. These capabilities could be implemented at the edge or in centralized cloud services, depending on the deployment scenario. A complementary effort involves the formalization of the CityLink vocabulary into a Web ontology and linked data context. Doing so would increase semantic expressiveness and improve interoperability, particularly when used in combination with semantic query engines and linked data discovery mechanisms.

This line of work could be extended through collaboration with the W3C Web of Things (WoT) Working Group. Such collaboration may lead to the proposal and implementation of new mechanisms and tools to define semantically rich and modular TMs. During the development of this dissertation, certain limitations were identified in the current WoT tooling for modeling and composing complex TMs in the envisioned ways. Addressing these limitations would not only strengthen the proposed framework but also contribute to the evolution and refinement of the WoT standard itself.

Looking ahead toward real-world deployments, the CityLink framework would benefit from validation on a larger-scale testbed. This would help identify weaknesses in the architecture and software components, ensuring the solution scales effectively and reliably under realistic conditions. As part of this process, further Embedded Core implementations could be developed using more resource-efficient programming languages and platforms, optimizing energy consumption and lowering the hardware requirements of end nodes.

Finally, while the WoT standard offers robust support for security mechanisms, this dissertation has not addressed this aspect due to scope and time constraints. Security remains a critical requirement in production environments, especially in city-scale deployments. Therefore, future work should prioritize the integration of authentication, secure onboarding, encrypted communication, and access control mechanisms throughout the CityLink architecture.

All in all, the proposed solution establishes a foundation for modular, adaptive, and semantically aware IoT systems that can evolve with the growing complexity of modern cities. As smart cities confront increasing challenges from climate change and unequal urbanization, the results are a promising indication of more adaptable and sustainable urban systems. Although Smart City

applications are envisioned as the primary use case, *CityLink*, as a general-purpose IoT framework, also holds potential for broader application across domains such as Industry 4.0 and smart agriculture, where heterogeneous device integration and interoperability are equally critical.

As a final remark, it is worth noting that the core ideas and architectural contributions presented in this dissertation have undergone external scientific validation. A peer-reviewed conference paper titled From *Fragmentation to Integration: A Framework for Heterogeneous IoT-based Smart City Systems* was accepted for presentation at the 30th IEEE International Conference on Emerging Technologies and Factory Automation, to be held in September 2025. This paper centers on the architectural model introduced in Chapter 4 and represents an early dissemination of the dissertation results to the broader academic and industrial community. Although the paper is not yet available in the IEEE digital library at the time of writing, its acceptance into a prestigious international conference reinforces the relevance and validity of this dissertation's contributions.

References

[1] A. S. Syed, D. Sierra-Sosa, A. Kumar, and A. Elmaghraby, "Iot in smart cities: A survey of technologies, practices and challenges," *Smart Cities 2021, Vol. 4, Pages 429-475*, vol. 4, pp. 429–475, 2 Mar. 2021, ISSN: 2624-6511. DOI: 10.3390/SMARTCITIES4020024.

[2] T. Singh, A. Solanki, S. K. Sharma, A. Nayyar, and A. Paul, "A decade review on smart cities: Paradigms, challenges and opportunities," *IEEE Access*, vol. 10, pp. 68 319–68 364, 2022, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3184710.

[3] A. Arora, A. Jain, D. Yadav, V. Hassija, V. Chamola, and B. Sikdar, "Next generation of multi-agent driven smart city applications and research paradigms," *IEEE Open Journal of the Communications Society*, vol. 4, pp. 2104–2121, 2023, ISSN: 2644125X. DOI: 10.1109/OJCOMS.2023.3310528.

[4] D. G. Costa and F. P. de Oliveira, "A prioritization approach for optimization of multiple concurrent sensing applications in smart cities," *Future Generation Computer Systems*, vol. 108, pp. 228–243, 2020, ISSN: 0167-739X. DOI: \url{https://doi.org/10.1016/j.future.2020.02.067}.

[5] J. C. N. Bittencourt, D. G. Costa, P. Portugal, and F. Vasques, "A survey on adaptive smart urban systems," *IEEE Access*, vol. 12, pp. 102 826–102 850, 2024, ISSN: 21693536. DOI: 10.1109/ACCESS.2024.3433381.

[6] M. Aly, F. Khomh, Y. G. Gueheneuc, H. Washizaki, and S. Yacout, "Is fragmentation a threat to the success of the internet of things?" *IEEE Internet of Things Journal*, vol. 6, pp. 472–487, 1 Feb. 2019, ISSN: 23274662. DOI: 10.1109/JIOT.2018.2863180.

[7] J. C. N. Bittencourt, D. G. Costa, P. Portugal, and F. Vasques, "Dynamic sensor configuration for multi-target emergency detection in smart cities," in *2023 IEEE International Smart Cities Conference (ISC2)*, 2023, pp. 1–4. DOI: 10.1109/ISC257844.2023.10293735.

[8] L. Sciullo, L. Gigli, F. Montori, A. Trotta, and M. D. Felice, "A survey on the web of things," *IEEE Access*, vol. 10, pp. 47 570–47 596, 2022, ISSN: 21693536. DOI: 10.1109/ACCESS.2022.3171575.

[9] CIM ETSI ISG, "Context Information Management (CIM); NGSI-LD API," ETSI, Group Specification, Mar. 2024.

[10] T. Ribeiro, J. P. Coelho, L. Jorge, J. Sardao, J. Goncalves, and H. Rosse, "Modelling with ngsi-ld: The vallpass project case study," in *IEEE International Conference on Industrial Informatics (INDIN)*, vol. 2023-July, 2023, ISBN: 9781665493130. DOI: 10.1109/INDIN51400.2023.10218110.

[11] F. Martella, V. Lukaj, M. Fazio, A. Celesti, and M. Villari, "On-demand and automatic deployment of microservice at the edge based on ngsi-ld," in *Proceedings - 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2023*, 2023, pp. 314–320, ISBN: 9798350337631. DOI: 10.1109/PDP59025.2023.00055.

[12] *oneM2M*, https://www.onem2m.org/, Accessed: 30-06-2025.

[13] L. Sciullo, C. Castiglione, A. Trotta, and M. D. Felice, "Wot on the extreme edge (wottee): Enabling w3c web of things for micro-controllers," in *2022 IEEE 8th World Forum on Internet of Things, WF-IoT 2022*, Institute of Electrical and Electronics Engineers Inc., 2022, ISBN: 9781665491532. DOI: 10.1109/WF-IoT54382.2022.10152179.

[14] D. Ibaseta, A. García, M. Álvarez, *et al.*, "Monitoring and control of energy consumption in buildings using wot: A novel approach for smart retrofit," *Sustainable Cities and Society*, vol. 65, Feb. 2021, ISSN: 22106707. DOI: 10.1016/j.scs.2020.102637.

[15] D. G. Costa, J. C. N. Bittencourt, F. Oliveira, J. P. J. Peixoto, and T. C. Jesus, "Achieving sustainable smart cities through geospatial data-driven approaches," *Sustainability 2024, Vol. 16, Page 640*, vol. 16, p. 640, 2 Jan. 2024, ISSN: 2071-1050. DOI: 10.3390/SU16020640.

[16] T. Singh, A. Solanki, S. K. Sharma, A. Nayyar, and A. Paul, "A decade review on smart cities: Paradigms, challenges and opportunities," *IEEE Access*, vol. 10, pp. 68 319–68 364, Jun. 2022, ISSN: 21693536. DOI: 10.1109/ACCESS.2022.3184710.

[17] G. Schreiber and Y. Raimond, "RDF 1.1 Primer," W3C, Working Group Note, Jun. 2013, Accessed: 30-06-2025.

[18] M. Duerst and M. Suignard, "Internationalized resource identifiers (iris)," IETF - Network Working Group, Request for Comments, Jan. 2005, Accessed: 30-06-2025.

[19] JSON-LD Working Group, "JSON-LD 1.1," W3C, Recommendation, Jul. 2020, Accessed: 30-06-2025.

[20] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform resource identifier (uri): Generic syntax*, https://datatracker.ietf.org/doc/html/rfc3986, Request for Comments, Accessed: 30-06-2025, Jan. 2005.

[21] R. Moats, *Urn syntax*, https://datatracker.ietf.org/doc/html/rfc2141, Request for Comments, Accessed: 30-06-2025, May 1997.

[22] P. Leach, M. Mealling, and R. Salz, *A universally unique identifier (uuid) urn namespace*, https://datatracker.ietf.org/doc/html/rfc4122, Request for Comments, Accessed: 30-06-2025, Jul. 2005.

[23] K. Davis, B. Peabody, and P. Leach, *Universally unique identifiers (uuids)*, https://datatracker.ietf.org/doc/html/rfc9562, Accessed: 30-06-2025.

[24] Apache Software Foundation, *Apache Jena*, https://jena.apache.org/, Accessed: 30-06-2025.

[25] *Fluree Core*, https://flur.ee/fluree-products/intelligent-database/, Accessed: 30-06-2025.

[26] SPARQL Working Group, "SPARQL 1.1 Overview," W3C, Recommendation, Mar. 2013, Accessed: 30-06-2025.

[27] K. G. Clark, "Rdf data access use cases and requirements," W3C, Working Draft, Mar. 2005, Accessed: 30-06-2025.

[28] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph, "OWL 2 Web Ontology Language Primer (Second Edition)," W3C, Recommendation, Dec. 2012, Accessed: 30-06-2025.

[29] W3C OWL Working Group, "OWL 2 Web Ontology Language Document Overview (Second Edition)," W3C, Recommendation, Dec. 2012, Accessed: 30-06-2025.

[30] M. Lagally, R. Matsukura, M. McCool, and K. Toumura, "Web of Things (WoT) Architecture 1.1," W3C, Recommendation, Dec. 2023, Accessed: 30-06-2025.

[31] S. Kaebisch, M. McCool, and E. Korkan, "Web of Things (WoT) Thing Description 1.1," W3C, Recommendation, Dec. 2023, Accessed: 30-06-2025.

[32] A. Cimmino, M. McCool, F. Tavakolizadeh, and K. Toumura, "Web of Things (WoT) Discovery," W3C, Recommendation, Dec. 2023, Accessed: 30-06-2025.

[33] M. Lagally and M. McCool, "Iot interoperability with w3c web of things," in *Proceedings - IEEE Consumer Communications and Networking Conference, CCNC*, Institute of Electrical and Electronics Engineers Inc., 2022. DOI: 10.1109/CCNC49033.2022.9700546.

[34] M. Lagally, B. Francis, M. McCool, R. Matsukura, S. Kaebisch, and T. Mizushima, "Web of things (wot) profile," W3C, Working Draft, Jan. 2023, Accessed: 30-06-2025.

[35] *Heart Project*, https://heartproject.eu, Accessed: 30-06-2025.

[36] *Smart santander*, https://www.smartsantander.eu/, Accessed: 30-06-2025.

[37] L. Sánchez, L. Muñoz, J. Galache, *et al.*, "Smartsantander: Iot experimentation over a smart city testbed," *Computer Networks*, Jan. 2013. DOI: 10.1016/j.bjp.2013.12.020.

[38] Y. Ji, K. Ok, and W. S. Choi, "Web of things based iot standard interworking test case: Demo abstract," in *Proceedings of the 5th Conference on Systems for Built Environments*, ser. BuildSys '18, Shenzen, China: Association for Computing Machinery, 2018, pp. 182–183, ISBN: 9781450359511. DOI: 10.1145/3276774.3281012. [Online]. Available: https://doi.org/10.1145/3276774.3281012.

[39] *Node wot*, https://github.com/eclipse-thingweb/node-wot, Accessed: 30-06-2025.

[40] *FIWARE Foundation*, https://www.fiware.org/, Accessed: 30-06-2025.

[41] FIWARE Fundation, *Smart data models*, https://smartdatamodels.org/, Accessed: 30-06-2025.

[42] FIWARE Fundation, *Fiware generic enablers catalogue*, https://www.fiware.org/catalogue/, Accessed: 30-06-2025.

[43] F. Martella, G. Parrino, G. Ciulla, *et al.*, "Virtual device model extending ngsi-ld for faas at the edge," in *Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021*, May 2021, pp. 660–667, ISBN: 9781728195865. DOI: 10.1109/CCGrid51090.2021.00079.

[44] J. Hwang, L. Nkenyereye, N. Sung, J. Kim, and J. Song, "Iot service slicing and task offloading for edge computing," *IEEE Internet of Things Journal*, vol. 8, pp. 11 526–11 547, 14 Jul. 2021, ISSN: 23274662. DOI: 10.1109/JIOT.2021.3052498.

[45] S. Böhm and G. Wirtz, "Cloud-edge orchestration for smart cities: A review of kubernetes-based orchestration architectures," *EAI Endorsed Transactions on Smart Cities*, vol. 6, e2, 18 May 2022. DOI: 10.4108/eetsc.v6i18.1197.

[46] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, IEEE, Oct. 2017, pp. 1–7, ISBN: 978-1-5386-3403-5. DOI: 10.1109/SysEng.2017.8088251. [Online]. Available: http://ieeexplore.ieee.org/document/8088251/.

[47] "Embserve: Embedded services for constrained devices," *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*, vol. 2023-April, 2023. DOI: 10.1109/WFCS57264.2023.10144123.

[48] P.-A. Champin and N. Lindström, "RDF 1.2 Primer," W3C, Group Draft Note, Apr. 2025, Accessed: 30-06-2025.

[49] Eclipse Foundation, *Eclipse ThingWeb*, https://github.com/eclipse-thingweb, Accessed: 30-06-2025.

[50] *MicroPython*, https://micropython.org/, Accessed: 30-06-2025.

[51] *Zephyr RTOS*, https://zephyrproject.org/, Accessed: 30-06-2025.

[52] T. Amorim, *CityLink*, https://github.com/les2feup/CityLink, Accessed: 30-06-2025.

[53] M. Koster, E. Korkan, and C. Aguzzi, "Web of things (wot) mqtt binding template," W3C, Working Draft, May 2025, Accessed: 30-06-2025.

[54] Rapberry Pi Foundation, *RP2040*, https://www.raspberrypi.com/products/rp2040/, Accessed: 30-06-2025.

[55] Eclipse Foundation, *Eclipse Mosquitto*, https://mosquitto.org/, Accessed: 30-06-2025.

[56] Rapberry Pi Foundation, *Raspberry Pi Pico W*, https://www.raspberrypi.com/products/raspberry-pi-pico/, Accessed: 30-06-2025.

[57] micropython-lib, *Micropython-umqtt.simple*, https://pypi.org/project/micropython-umqtt.simple/, Accessed: 30-06-2025.

[58] Deno Team, *Deno, the next generation javascript runtime*, https://deno.com/, Accessed: 30-06-2025.

[59] MQTTjs Team, *Mqtt.js*, https://www.npmjs.com/package/mqtt, Accessed: 30-06-2025.

[60] Deno Team, *Fresh, the simple, approachable, productive web framework*, https://fresh.deno.dev/, Accessed: 30-06-2025.

[61] S. Gössner and G. Normington and C. Bormann, *Rfc 9535 jsonpath: Query expressions for json*, https://www.rfc-editor.org/rfc/rfc9535.html, Accessed: 30-06-2025.

[62] J. Robie, M. Dyck, and J. Spiegel, "XML Path Language (XPath) 3.1," W3C, Recommendation, Mar. 2017, Accessed: 30-06-2025.

[63] Preact Team, *React, fast 3kb alternative to react with the same modern api*, https://preactjs.com/, Accessed: 30-06-2025.

[64] Andrew Leech and Damien George, *Micropython cross compiler utility*, https://micropython.org/download/RPI_PICO_W/, Accessed: 30-06-2025.

[65] *Micropython port for rpi pico w*, https://micropython.org/download/RPI_PICO_W/, Accessed: 30-06-2025.

[66] *The Devicetree Specification*, https://www.devicetree.org, Accessed: 30-06-2025.

Appendix A

Thing Models

Listing A.1: Example of a complete *Application TM*

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        "https://raw.githubusercontent.com/w3c/wot-binding-templates/refs/heads/main/bindings/protocols/mqtt/context.jsonld",
        "https://raw.githubusercontent.com/les2feup/CityLink/refs/heads/main/context.jsonld"
    ],
    "@type": [
        "tm:ThingModel",
        "citylink:AppTM"
    ],
    "title": "CityLink Light Control",
    "description": "A simple application to control a light bulb.",
    "version": "0.1.0",
    "links": [
        {
            "rel": "tm:submodel",
            "href": "https://example.org/citylink/tm/platform/smart-light-bulb.tm.json",
            "type": "application/tm+json",
            "instanceName": "citylink:platform"
        },
        {
            "rel": "tm:submodel",
```

```
            "href": "https://example.org/citylink/tm/embedded-core/micropython-mqtt.tm.json",
            "type": "application/tm+json",
            "instanceName": "citylink:embeddedCore"
        }
    ],
    "manifest": {
        "placeholder": {
            "SOME_PLACEHOLDER": "some-value"
        },
        "source": [
            {
                "filepath": "main.py",
                "href": "https://example.org/citylink/application/citylink-light-control/source/main.py",
                "contentType": "text/x-python",
                "sha256": "{sha256-hash-of-the-file}"
            }
        ]
    },
    "properties": {
        "status": {
            "type": "boolean",
            "description": "The current status of the light bulb.",
            "readOnly": true,
            "writeOnly": false,
            "observable": true,
            "forms": [
                {
                    "href": "mqtt://broker.example.org:1883",
                    "mqv:filter":
                     "citylink/{{NodeID}}/light/status",
                    "mqv:qos": 0,
                    "mqv:retain": true,
                    "contentType": "application/json",
                    "op": [
                        "readproperty",
                        "observeproperty",
                        "unobserveproperty"
                    ]
```

```
60                  }
61              ]
62          }
63      },
64      "actions": {
65          "toggle": {
66              "description": "Toggle the light bulb on or off.",
67              "input": {
68                  "type": "boolean",
69                  "title": "Toggle State",
70                  "description": "The desired state of the light bulb.
                     ↪  'true' for on, 'false' for off."
71              },
72              "forms": [
73                  {
74                      "href": "mqtt://broker.example.org:1883",
75                      "mqv:topic": "citylink/{{NodeID}}/light/toggle",
76                      "mqv:qos": 0,
77                      "mqv:retain": false,
78                      "contentType": "application/json",
79                      "op": "invokeaction"
80                  }
81              ]
82          }
83      },
84      "events": {
85          "energyAlert": {
86              "title": "Energy Consumption Alert",
87              "description": "An alert event that is triggered if the
                 ↪  light is on for too long.",
88              "data": {
89                  "type": "object",
90                  "properties": {
91                      "timestamp": {
92                          "type": "string",
93                          "format": "date-time",
94                          "description": "The time when the alert was
                             ↪  triggered."
95                      },
96                      "turnOnTime": {
```

```
                            "type": "string",
                            "format": "date-time",
                            "description": "The time when the light was
                             ↪ turned on."
                        },
                        "duration": {
                            "type": "integer",
                            "description": "The duration in seconds that
                             ↪ the light has been on."
                        }
                    }
                },
                "forms": [
                    {
                        "href": "mqtt://broker.example.org:1883",
                        "mqv:filter": "citylink/{{NodeID}}/light/alert",
                        "mqv:qos": 0,
                        "mqv:retain": false,
                        "contentType": "application/json",
                        "op": "subscribeevent"
                    }
                ]
            }
        }
}
```

Listing A.2: Example of a complete protocol-agnostic Application Thing Model

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        "https://raw.githubusercontent.com/les2feup/CityLink/refs/h
        ↪ eads/main/context.jsonld"
    ],
    "@type": [
        "tm:ThingModel",
        "citylink:AppTM"
    ],
    "title": "CityLink Light Control",
    "description": "A simple application to control a light bulb.",
    "version": "0.1.0",
    "links": [
        {
            "rel": "tm:submodel",
            "href": "https://example.org/citylink/tm/platform/smart
            ↪ -light-bulb.tm.json",
            "type": "application/tm+json",
            "instanceName": "citylink:platform"
        },
        {
            "rel": "tm:submodel",
            "href": "https://example.org/citylink/tm/embedded-core/
            ↪ micropython-base.tm.json",
            "type": "application/tm+json",
            "instanceName": "citylink:embeddedCore"
        },
        {
            "rel": "citylink:manifestLink",
            "href": "https://example.org/citylink/application/light
            ↪ -control/manifest.json",
            "type": "application/json"
        }
    ],
    "properties": {
        "status": {
```

```
            "type": "boolean",
            "description": "The current status of the light bulb.",
            "readOnly": true,
            "writeOnly": false,
            "observable": true
        }
    },
    "actions": {
        "toggle": {
            "description": "Toggle the light bulb on or off.",
            "input": {
                "type": "boolean",
                "title": "Toggle State",
                "description": "The desired state of the light bulb.
                 ↪ 'true' for on, 'false' for off."
            }
        }
    },
    "events": {
        "energyAlert": {
            "title": "Energy Consumption Alert",
            "description": "An alert event that is triggered if the
             ↪ light is on for too long.",
            "data": {
                "type": "object",
                "properties": {
                    "timestamp": {
                        "type": "string",
                        "format": "date-time",
                        "description": "The time when the alert was
                         ↪ triggered."
                    },
                    "turnOnTime": {
                        "type": "string",
                        "format": "date-time",
                        "description": "The time when the light was
                         ↪ turned on."
                    },
                    "duration": {
                        "type": "integer",
```

```
70                              "description": "The duration in seconds that
                                ↪  the light has been on."
71                          }
72                      }
73                  }
74              }
75          }
76      }
```

Listing A.3: Example of a protocol-specific *Application TM*

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        "https://example.org/CityLink/context.jsonld"
    ],
    "@type": [ "tm:ThingModel", "citylink:AppTM" ],
    "name": "MQTT CityLink Light Control",
    "links": [
        {
            "rel": "tm:submodel",
            "href": "https://example.org/citylink/tm/embedded-core/micropython-mqtt.tm.json",
            "type": "application/tm+json",
            "instanceName": "citylink:embeddedCore"
        },
        {
            "rel": "tm:extends",
            "href": "https://example.org/citylink/tm/application/citylink-light-control-base.tm.json",
            "type": "application/tm+json",
        }
    ],
    "properties": {
        "status": {
            "forms": [
                {
                    "href": "mqtt://broker.example.org:1883",
                    "mqv:filter":
                     "citylink/{{NodeID}}/light/status",
                    "mqv:qos": 0,
                    "mqv:retain": true,
                    "contentType": "application/json",
                    "op": [ "readproperty", "observeproperty",
                     "unobserveproperty" ]
                }
            ]
        }
```

```
    },
    "actions": {
        "toggle": {
            "forms": [
                {
                    "href": "mqtt://broker.example.org:1883",
                    "mqv:topic": "citylink/{{NodeID}}/light/toggle",
                    "mqv:qos": 0,
                    "mqv:retain": false,
                    "contentType": "application/json",
                    "op": "invokeaction"
                }
            ]
        }
    },
    "events": {
        "energyAlert": {
            "forms": [
                {
                    "href": "mqtt://broker.example.org:1883",
                    "mqv:filter": "citylink/{{NodeID}}/light/alert",
                    "mqv:qos": 0,
                    "mqv:retain": false,
                    "contentType": "application/json",
                    "op": "subscribeevent"
                }
            ]
        }
    }
}
```

Listing A.4: *Application TM* for the Example Application

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        "https://raw.githubusercontent.com/les2feup/CityLink/refs/h
        ↪ eads/main/context.jsonld"
    ],
    "@type": [ "tm:ThingModel", "citylink:AppTM" ],
    "title": "Simple Temperature Sensor",
    "description": "Simple temperature sensor with alarm
    ↪ functionality. Base application model meant to be extended
    ↪ by concrete implementations.",
    "version": {
        "model": "0.1.0"
    },
    "properties": {
        "temperature": {
            "title": "Temperature sample",
            "description": "The current temperature value in degrees
            ↪ Celsius",
            "type": "integer",
            "readOnly": true,
            "writeOnly": false
        }
    },
    "events": {
        "overheating": {
            "title": "Overheating event",
            "description": "Triggered when the temperature exceeds a
            ↪ certain threshold",
            "data": {
                "type": "object",
                "properties": {
                    "timestamp": {
                        "type": "string",
                        "format": "date-time",
                        "description": "The time when the event was
                        ↪ triggered"
```

```
                    },
                    "alarm": {
                        "type": "boolean",
                        "description": "Indicates if the alarm is
                        ↪ active"
                    },
                    "temperature": {
                        "type": "integer",
                        "description": "The temperature value that
                        ↪ triggered the event"
                    }
                }
            }
        },
        "actions": {
            "toggleAlarm": {
                "title": "Toggle alarm",
                "description": "Toggle the alarm state",
                "input": {
                    "type": "boolean",
                    "description": "Alarm state (on/off) to set"
                }
            }
        }
}
```

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        "https://raw.githubusercontent.com/les2feup/CityLink/refs/h
         ↪ eads/main/context.jsonld"
    ],
    "@type": [
        "tm:ThingModel",
        "citylink:PlatTM"
    ],
    "title": "Raspberry Pi Pico W",
    "description": "Thing Model for the Raspberry Pi Pico W.",
    "version": {
        "model": "0.1.0"
    },
    "links": [
        {
            "rel": "tm:submodel",
            "href": "https://raw.githubusercontent.com/les2feup/Cit
             ↪ yLink/refs/heads/main/ThingModels/platform/rp2040.t
             ↪ m.json",
            "type": "application/tm+json",
            "instanceName": "MCU"
        },
        {
            "rel": "tm:submodel",
            "href": "https://raw.githubusercontent.com/les2feup/Cit
             ↪ yLink/refs/heads/main/ThingModels/platform/infineon
             ↪ _cyw43439.tm.json",
            "type": "application/tm+json",
            "instanceName": "NIC"
        }
    ],
    "flash": {
        "tm:ref": "https://raw.githubusercontent.com/les2feup/CityL
         ↪ ink/refs/heads/main/ThingModels/platform/rp2040.tm.json
         ↪ #/properties/flash",
        "properties": {
            "size": {
                "const": 2
            }
        }
    }
}
```

Listing A.5: *Platform TM* for the Raspberry Pi Pico W

```
{
    "actions": {
        "registration": {
            "title": "End Node Registration",
            "description": "Register a new end node with the
             ↪  connector.",
            "input": {
                "type": "object",
                "description": "Input for registering an end node
                 ↪  with the connector.",
                "properties": {
                    "tm": {
                        "type": "string",
                        "description": "URL of the Thing Model
                         ↪  describing the end node."
                    },
                    "placeholder": {
                        "type": "object",
                        "description": "Key-value pairs for
                         ↪  additional parameters required for
                         ↪  registration.",
                    }
                },
                "required": [ "tm" ]
            },
            "forms": [
                {
                    "href": "{{CITYLINK_HREF}}",
                    "mqv:topic": "citylink/{nodeID}/registration",
                    "contentType": "application/json",
                    "op": "invokeaction",
                    "mqv:qos": 2,
                    "mqv:retain": false
                }
            ]
        }
    }
}
```

Listing A.6: End Node Registration Action

Listing A.7: End Node Registration Event

```
{
  "events": {
    "response": {
      "title": "End Node Registration Response",
      "description": "Event triggered when an end node registration
       ↪  is acknowledged by the controller.",
      "data": {
        "oneOf": [
          {
            "type": "object",
            "title": "Registration Acknowledgment",
            "description": "Acknowledgment of the end node
             ↪  registration. Signaling the start of the
             ↪  registration procedure.",
            "properties": {
              "status": {
                "type": "string",
                "const": "ack"
              }
            }
          },
          {
            "type": "object",
            "title": "Registration Error",
            "description": "Error during the end node registration
             ↪  procedure.",
            "properties": {
              "status": {
                "type": "string",
                "const": "error"
              },
              "message": {
                "type": "string",
                "description": "Description of the error that
                 ↪  occurred during registration."
              }
            }
```

```
            },
            {
              "type": "object",
              "title": "Registration Success",
              "description": "Successful end node registration.",
              "properties": {
                "status": {
                  "type": "string",
                  "const": "success"
                },
                "id": {
                  "type": "string",
                  "description": "Unique identifier assigned to the
                   ↪  end node after successful registration."
                }
              }
            }
          ]
        },
        "forms": [
          {
            "href": "{{CITYLINK_HREF}}",
            "mqv:topic": "citylink/{nodeID}/registration/ack",
            "contentType": "application/json",
            "op": "subscribeevent",
            "mqv:qos": 2,
            "mqv:retain": false
          }
        ]
      }
    }
}
```

Appendix B

Embedded Core Micropython Application API

B.1 Affordance Handlers

```
# general api
def create_property(self, property_name, initial_value, **_): ...

# uMQTT core
def create_property(self, property_name, initial_value,
 ↪  pub_only=True, **_): ...
```

Create a property with an initial value. '**_' is used to ignore all additional keyword arguments.

The uMQTT core variant allows for a 'pub_only' argument, which indicates that the property should not be writable by consumers. If set to 'True', the uMQTT core will register a setter method, allowing consumers to write the property value by publishing to the

'citylink/{node_id}/actions/set/{property_name}' topic.

```
def get_property(self, property_name, **_): ...
```

Retrieve the value of a property by its name. '**_' is used to ignore all additional keyword arguments.

```
# general api
def set_property(self, property_name, value, **_): ...

# uMQTT core
def set_property(self, property_name, value, retain=True, qos=0,
 ↪  **_): ...
```

Set the value of a property by its name. '**_' is used to ignore all additional keyword arguments.

The general API variant does not support additional arguments, while the uMQTT core variant allows for 'retain' and 'qos' parameters to control MQTT message properties.

```
# general api
def emit_event(self, event_name, event_data, **_): ...

# uMQTT core
def emit_event(self, event_name, event_data,
               retain=False, qos=0, **_): ...
```

Emit an event with a name and data. '**_' is used to ignore all additional keyword arguments.

The general API variant does not support additional arguments, while the uMQTT core variant allows for 'retain' and 'qos' parameters to control MQTT message properties.

```
# general api
def register_action_executor(self, action_name, action_func, **_):
 ↪  ...

# uMQTT core
def register_action_executor(self, action_name, action_func, qos=0,
 ↪  **_): ...
```

Register an action executor for a given action name. '**_' is used to ignore all additional keyword arguments.

The general API variant does not support additional arguments, while the uMQTT core variant allows for a 'qos' parameter to control the subscription quality of service.

```
def sync_executor(func): ...
```

This decorator is used to wrap a synchronous function so that it can be executed by the asynchronous event loop. Used to mark task functions and action handlers that are meant to be invoked by the Embedded Core and not by the application code directly.

```
def async_executor(func): ...
```

This decorator is used to wrap an asynchronous function so that it can be executed by the asynchronous event loop. Used to mark task functions and action handlers that are meant to be invoked by the Embedded Core and not by the application code directly.

B.2 Task Scheduling

```
def task_create(self, task_id, task_func, period_ms=0): ...
```

Create a task with a unique identifier and a function to execute. The ‘period_ms‘ argument specifies the period in milliseconds at which the task should be executed. If set to 0, the task will only run once.

```
def task_cancel(self, task_id): ...
```

Cancel a running task by its unique identifier. This will also remove the task from the task list.

```
async def task_sleep_s(self, s): ...
```

Sleep for a specified number of seconds in an asynchronous context. Use inside an asynchronous function to pause execution without blocking the event loop.

```
async def task_sleep_ms(self, ms): ...
```

Sleep for a specified number of milliseconds in an asynchronous context. Use inside an asynchronous function to pause execution without blocking the event loop.

Appendix C

Code Snippets

```python
from citylink.core import EmbeddedCore
from micropython import const
from machine import Pin, ADC
from time import time, gmtime

temp_sensor = ADC(Pin(26))
alarm = Pin(16, Pin.OUT)

@EmbeddedCore.sync_executor
def toggle_alarm(core: EmbeddedCore, state: bool):
    alarm.value(1 if state else 0)

@EmbeddedCore.sync_executor
def sample_temp(core: EmbeddedCore):
    raw_value = temp_sensor.read_u16()  # raw 0-65535 ADC reading
    temperature = (raw_value / 65535) * 100 # scaled 0-100
    if temperature > 40:
        core.emit_event( "overheating", {
                "timestamp": gmtime(time()),
                "alarm": alarm.value(),
                "temperature": temperature,
                })
    core.set_property("temperature", temperature)

def setup(core: EmbeddedCore):
    core.create_property("temperature", 0.0)
    core.register_action_executor("toggleAlarm", toggle_alarm)
    core.task_create( "sample_temp", sample_temp, period_ms=1000)
```

Listing C.1: Example Application using the uMQTT Core API

```typescript
import { ThingDirectory } from "@citylink-edgenode/thing-directory";
import { MqttEdgeConnector } from
  ↪ "@citylink-edgenode/connector-mqtt";
import {
  UMQTTCoreControllerFactory,
} from "@citylink-edgenode/controller-umqtt-core";
import * as cl from "@citylink-edgenode/core";

const loadTM = async (path: string): Promise<cl.ThingModel> => {
  const decoder = new TextDecoder("utf-8");
  const file = decoder.decode(await Deno.readFile(path));
  return JSON.parse(file) as cl.ThingModel;
};

const controllerTM: cl.ThingModel = await loadTM(
  "../../ThingModels/controllers/mqtt-mpy-core-controller.tm.json",
);
const edgeConTM: cl.ThingModel = await loadTM(
  "../../ThingModels/connectors/edge-connector-mqtt.tm.json",
);

const tdOpts: cl.ThingDescriptionOpts = {
  placeholderMap: {
    CITYLINK_ID: `urn:uuid:${crypto.randomUUID()}`,
    CITYLINK_HREF: "mqtt://localhost:1883",
  },
  selfComposition: true,
};

const edgeConTD: cl.ThingDescription = await cl.utils.produceTD(
  edgeConTM,
  tdOpts,
);

const mqttConnector = new MqttEdgeConnector(edgeConTD);
mqttConnector.registerControllerFactory(
  new UMQTTCoreControllerFactory(controllerTM),
);

await mqttConnector.startRegistrationListener();

const thingDirectory = new ThingDirectory();
thingDirectory.addEdgeConnector(mqttConnector);
thingDirectory.start();
```

Listing C.2: Edge Connector deployment script

Appendix D

uMQTT Core Adaptation Affordances

This appendix describes the MQTT-based WoT affordances exposed by the *uMQTT Core* runtime to support over-the-air (OTA) adaptation of end nodes. These affordances include actions for triggering adaptation, transferring files, and handling completion or rollback, as well as events for reporting results.

All MQTT topics referenced below are prefixed with `citylink/{node_id}/`, where `{node_id}` is a unique identifier assigned during registration.

D.0.0.1 OTAUInit Action

Purpose. Signals the end node to enter the *Adaptation* state. This results in a reboot into a minimal runtime capable of receiving updates.

Topic. `citylink/{node_id}/actions/core/otau/init`

Input.
```
tm_url: string # URL pointing to the new Application TM
```

D.0.0.2 OTAUFinish Action

Purpose. Signals the end node to complete the adaptation. If the commit flag is set to *true*, the node applies the changes and reboots. If *false*, it discards the updates and restores its previous state.

Topic. `citylink/{node_id}/actions/core/otau/finish`

Input.
```
commit: bool # Whether to commit (true) or discard (false) the s
taged changes.
```

D.0.0.3 OTAUWrite Action

Purpose. Transfers a file to the end node's virtual file system. The file is buffered with a '.temp' suffix and verified using a hash.

Topic. `citylink/{node_id}/actions/core/otau/write`

Input.

```
path: str  # Target file path (relative to root)
payload: {
    "data": str,    # Base64-encoded file contents
    "hash": str,    # Hash of encoded contents
    "algo": str,    # Hash algorithm (e.g., "crc32")
}
append: bool | None  # Optional; whether to append or overwrite
```

D.0.0.4 OTAUDelete Action

Purpose. Deletes a file or directory from the end node's virtual file system. The deletion is buffered until the commit step.

Topic. `actions/core/otau/delete`

Input.

```
path: str        # Target file path (relative to root)
recursive: bool  # Optional; if true, deletes recursively
```

D.0.0.5 OTAUResult Event

Purpose. Emitted by the end node to report the result of an adaptation action (write or delete). Allows the edge node to monitor progress and handle errors.

Topic. `citylink/{node_id}/events/core/otau/report`

Data.

```
timestamp: {
    "epoch_year": int | None,   # Optional; base year (default 1970)
    "seconds": int              # Seconds since epoch
},
result: {
    "error": bool,                # True if operation failed
    "written": str | None,        # Path written, if any
    "deleted": list[str]          # Paths queued for deletion
}
```

Appendix E

Web Interface Screenshots

.

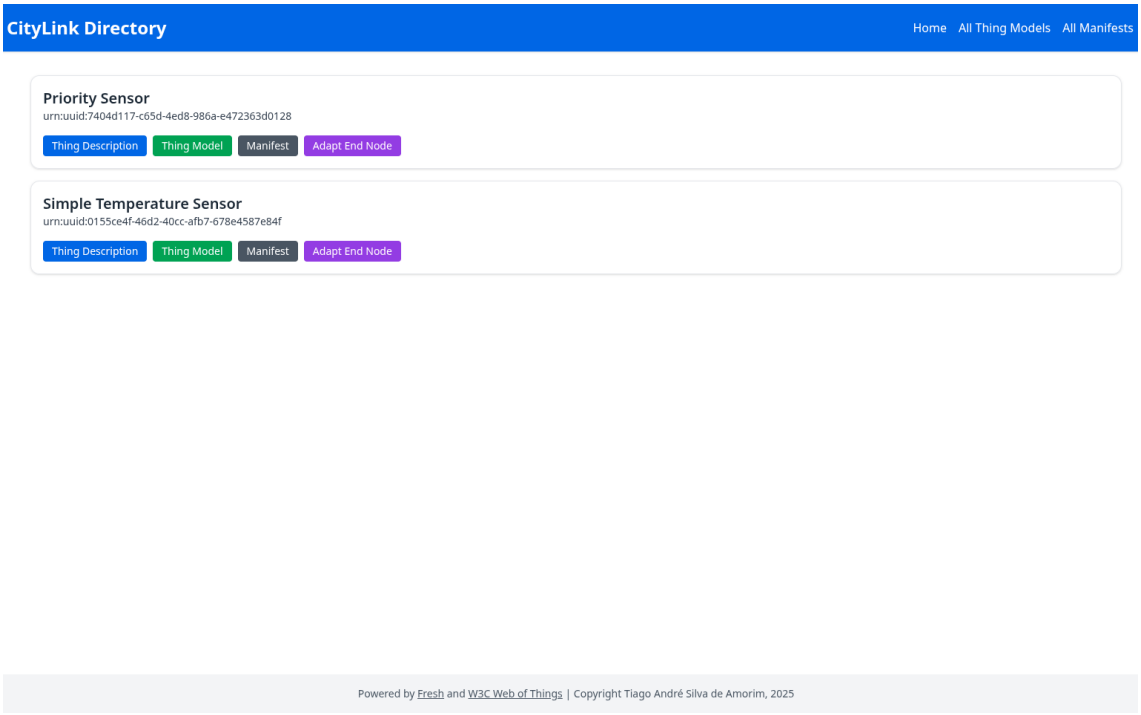


Figure E.1: Web Interface Home Page Screenshot.

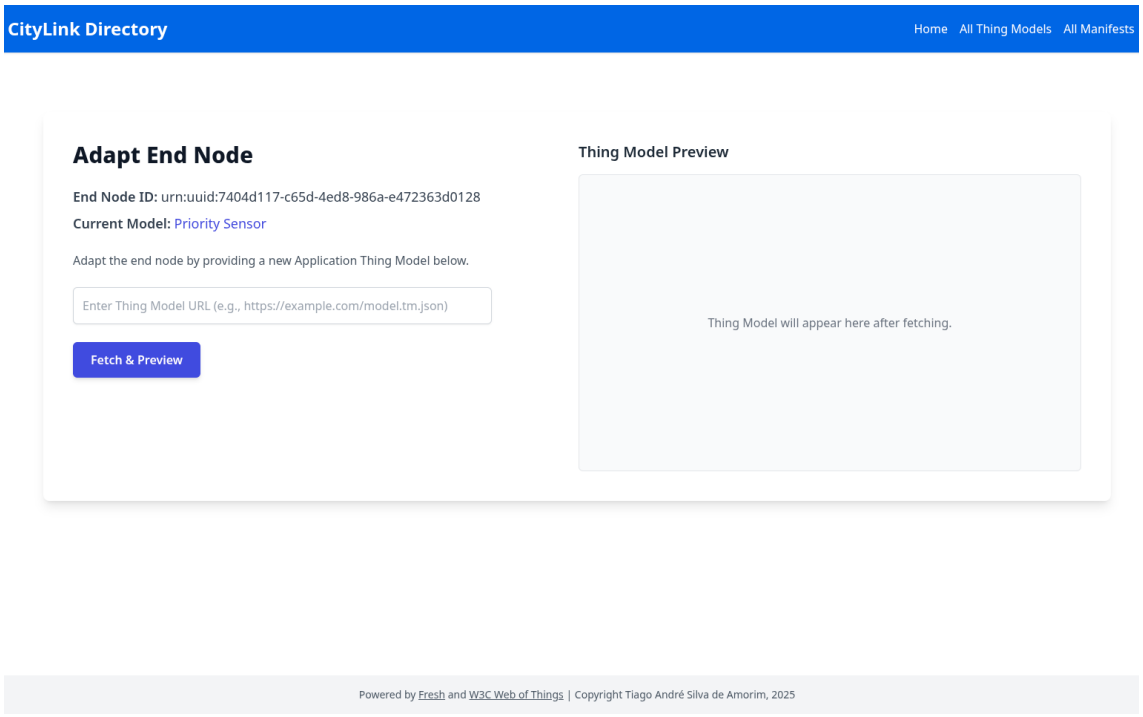


Figure E.2: Web Interface End Node Adaptation page.

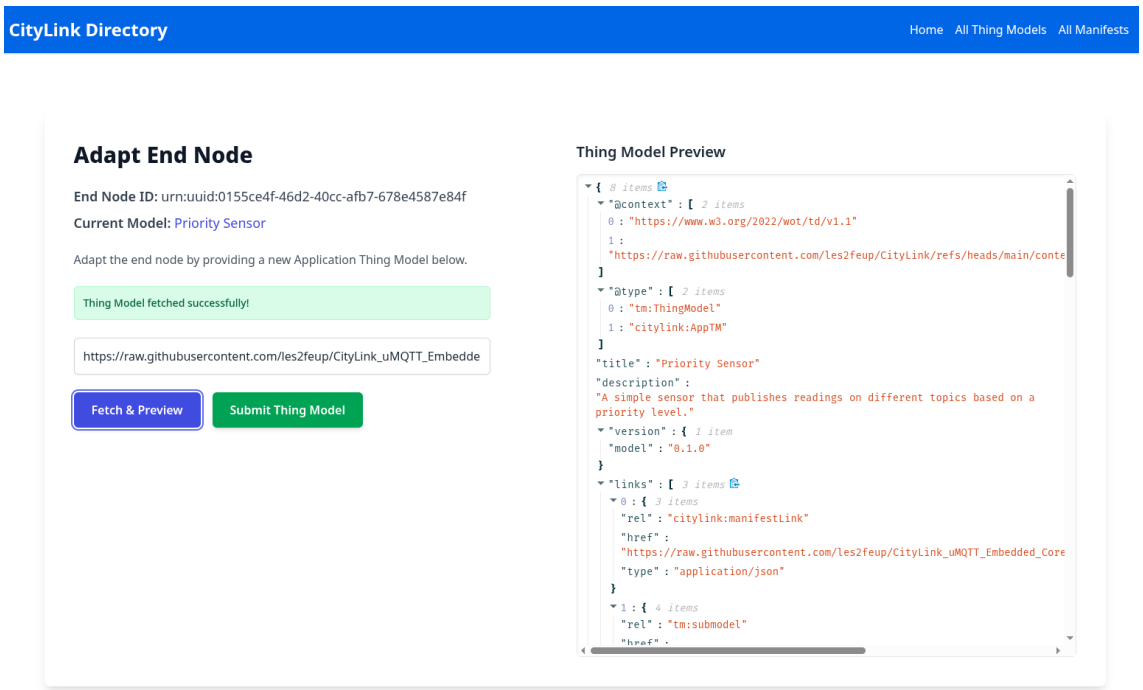


Figure E.3: Web Interface End Node Adaptation page, with TM preview.

Figure E.4: Web Interface ‘priority‘ Property Affordance detailed view.

Figure E.5: Web Interface ‘sensor_value‘ Event Affordance detailed view.

Appendix F

Accepted Paper

A research paper incorporating contributions from this dissertation was accepted for publication at the 2025 IEEE ETFA (Emerging Technologies and Factory Automation) Conference. The full paper is included in the proceedings of the 30th edition of IEEE ETFA, while partial content is presented in this appendix.

Paper Title

From Fragmentation to Integration: A Framework for Heterogeneous IoT-based Smart City Systems

Authors

Tiago A. Amorim[1], João Carlos N. Bittencourt[2,3], Daniel G. Costa[4], Paulo Portugal[4]

[1] M.EEC, Faculty of Engineering, University of Porto, Portugal

[2] CETEC, Federal University of Recôncavo da Bahia, Brazil

[3] PDEEC, Faculty of Engineering, University of Porto, Portugal

[4] SYSTEC-ARISE, Faculty of Engineering, University of Porto, Portugal

Contact Emails

- tiago.andre.amorim@gmail.com
- joaocarlos@ufrb.edu.br
- danielgcosta@fe.up.pt
- pportugal@fe.up.pt

Acknowledgments

This work was supported by the Associate Laboratory Advanced Production and Intelligent Systems – ARISE LA/P/0112/2020 (DOI: 10.54499/LA/P/0112/2020), by the Base Funding (UIDB/00147/2020) and Programmatic Funding (UIDP/00147/2020) of the R&D Unit Center for Systems and Technologies – SYSTEC, and by the Fundação para a Ciência e a Tecnologia (FCT) through the PhD scholarship (2024.02446.BD).

Abstract

The development of data-driven smart cities has been primarily supported by the Internet of Things (IoT) paradigm, with sensors and actuators playing a critical role in a myriad of applications. However, as IoT Fragmentation becomes a reality due to diverse hardware and networking standard settings, interoperability among heterogeneous IoT devices remains a persistent challenge. This paper proposes a holistic approach for hardware interoperability on the edge layer, leveraging the W3C Web of Things (WoT) standard as a reference. A new system framework and a suite of software components for edge and end nodes enable operational services such as self-identification, dynamic over-the-air reconfiguration, and automatic device onboarding. By doing so, our approach facilitates seamless integration across heterogeneous hardware and communication protocols while maintaining semantic consistency through WoT-based data modeling, potentially contributing to the easier and faster development of smart city applications.

Keywords

Hardware interoperability, Edge computing, Wireless sensor networks, Adaptable sensors, Semantic sensor networks, Linked data.

Conclusions

This paper presents a holistic solution for enhancing interoperability and adaptability at the edge layer of large-scale IoT deployments, particularly in smart city environments. Motivated by the challenges of IoT fragmentation and the necessity for flexible, adaptive urban infrastructures, the proposed solution utilizes the W3C Web of Things standard and semantic web technologies to establish a consistent, machine-readable interface for devices within a monitoring infrastructure.

The proposed solution includes software components that allow end-node hardware requirements to be exchanged interchangeably, best fitting various deployment scenarios. Another key contribution is the *Thing Model* framework, which integrates with the software elements to enable automatic device registration, over-the-air updates, and seamless integration of heterogeneous platforms—all described through semantically rich descriptions enabled by the WoT standard.

A proof of concept demonstrates the practicality of the proposed solution using MicroPython and MQTT. It showcases automatic registration, device mutation, and simplified development workflows through reusable application models and rich APIs. This implementation validates the feasibility of the approach with lightweight, low-power devices and open-source toolchains.

Based on the preliminary results, we draw several directions for future work. One path is the formal definition and structuring of *Platform TMs*. This could be inspired by the *DeviceTree* specification [66], which has been extensively used by projects like the Linux kernel and Zephyr RTOS to model device hardware and support code generation. Structuring *Platform TMs* as a linked data representation of *DeviceTree* structures could enable advanced code generation features to be supported by the proposed framework.

Additionally, mechanisms for automatic edge node adaptation based on the devices in their vicinity could enhance the network's resilience and efficiency. Finally, integrating semantic search engines and linked data stores (e.g., RDF databases with SPARQL querying) into the edge node feature set or a centralized cloud service could unlock advanced capabilities for discoverability and orchestration of the end node infrastructure in smart urban environments.

The proposed solution establishes a foundation for modular, adaptive, and semantically aware IoT systems that can evolve with the growing complexity of modern cities. As smart cities confront increasing challenges from climate change and unequal urbanization, the results are a promising indication of more adaptable and sustainable urban systems.